Efficient Conditional Synchronization for Transactional Memory Based System

A Thesis
Presented to
The Academic Faculty

by

Aniket Naik

In Partial Fulfillment
of the Requirements for the Degree
Master of Science

School of Electrical and Computer Engineering
Georgia Institute of Technology
May 2006

Efficient Conditional Synchronization for Transactional Memory Based System

Approved by:

Dr. Hsien-Hsin S. Lee, Advisor
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Milos Prvulovic
College of Computing
Georgia Institute of Technology

Dr. Sudhakar Yalamanchili
School of Electrical and Computer Engineering
Georgia Institute of Technology

Date Approved April 4^{th}, 2006

To my family,

for their unwavering support,

in all my endeavours.

ACKNOWLEDGEMENTS

This dissertation would not have been possible without the people who encouraged me and provided me with their professional guidance and personal support. First of all, I would like to thank my parents, Dilip and Moti Naik,and my brother Amit Naik for the support and inspiration the provided every step of the way. I would also like to extend my heartfelt thanks to Prof. Milos Prvulovic, whose professional guidance was invaluable in realizing this work. He has been extremely patient with me and answered every question I ever asked.

I would like to thank members of my thesis reading committee, Prof. Lee and Prof. Yalamanchili for their valuable feedback on my thesis.

Lastly, I would like to thank, all members of my research group, especially, Chenyu Yan for providing important inputs to this work.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Multi-threaded applications are needed to realize the full potential of new chip-multi-threaded machines. Such applications are very difficult to program and orchestrate correctly, and transactional memory has been proposed as a way of alleviating some of the programming difficulties. However, transactional memory can directly be applied only to critical sections, while conditional synchronization remains difficult to implement correctly and efficiently.

The main contribution of the work presented in this dissertation is EasySync, a simple and inexpensive extension to transactional memory that allows arbitrary conditional synchronization to be expressed in a simple and composable way. Transactional memory eliminates the need to use locks and provides composability for critical sections: atomicity of a transaction is guaranteed regardless of how other code is written. EasySync provides the same benefits for conditional synchronizations: it eliminates the need to use conditional variables, and it guarantees wake-up of the waiting transaction when the real condition it is waiting for is satisfied, regardless of whether other code correctly signals that change. EasySync also allows transactional memory systems to efficiently provide lock-free and condition variable-free conditional critical regions and even more advanced synchronization primitives, such as guarded execution with arbitrary conditional or guard code.

Because EasySync informs the hardware the that a thread is waiting, it allows simple and effective optimizations, such as stopping the execution of a thread until there is a change in the condition it is waiting for. Like transactional memory, EasySync is backward-compatible with existing code, which we confirm by running unmodified Splash-2 applications linked with an EasySync-based synchronization library. We also re-write some of the synchronization in three Splash-2 applications, to take advantage of better code readability, and to replace spin-waiting with its more efficient EasySync equivalents.

Other contributions of this work are regarding the software impact of transactions in

general and EasySync in particular. A detailed study of effect on programmability when targeting hardware with EasySync and transactional memory is performed. In addition the algorithmic impact of EasySync is also considered and guidelines for developing parallel programs are suggested.

Our experimental evaluation shows that EasySync successfully eliminates processor activity while waiting, reducing the number of executed instructions by 8.6% on average in a 16-processor CMP. We also show that these savings increase with the number of processors, and also for applications written for transactional memory systems. Finally, EasySync imposes virtually no performance overheads, and can in fact improve performance.

CHAPTER I

INTRODUCTION

Traditionally, general purpose processors have been single core chips meant to provide optimal single-threaded performance. However, improving technology and reducing feature size coupled with increasing transistor densities, have led to the development of multi-core processor chips which are rapidly taking over the market for general-purpose processors. In order to realize the full potential of these new processors, parallel (multi-threaded) software programs will be required. However, the cost of developing such programs increases rapidly as their complexity and use increases.

While the complexity of writing a high-performance single-thread program is considerable, it is significantly higher for multi-threaded programs [7, 28]. Threads signify instruction sequences that can be executed concurrently. Even though the use of threads simplifies the conceptual design of programs, care and expertise is required to coordinate correct interaction among various threads. This is primarily because significant complexity is added while orchestrating access to shared data objects which require complicated reasoning. Synchronization mechanisms are used to correctly coordinate thread accesses to shared objects. These mechanisms often enforce some form of serialization to maintain a globally consistent view of shared data. While conservative use of such mechanisms aids in correct programming, they unnecessarily enforce serialization of thread execution and consequently degrade performance. On the other hand, omitting necessary synchronization leads to non deterministic behavior and may also lead to deadlocks. Another aspect of synchronization is energy efficiency.Even though programs with the optimal amount of synchronization may achieve high performance and be error-free, slight imbalances in load may significant degrade the energy efficiency of the program. Writing correct, high-performance, and energy-efficient multi-threaded programs thus entails a careful trade-off among various aspects of the program. These aspects include the ease of writing a correct program, its performance, and the

energy expended, especially during synchronization. Before we discuss the above aspects, we briefly discuss some of the popular software and hardware mechanisms for coordinating concurrent access to shared data.

1.1 Synchronization Mechanisms

Synchronization mechanisms are primarily required to coordinate access to shared data. The most popular constructs used for synchronization are critical sections.Critical sections are regions of code in which only one thread is allowed to operate on the object at any given time. This allows programmers to trivially satisfy serializability and enforce mutually exclusive access among threads to shared objects. Locks (also called mutexes) are the most commonly used primitives to implement critical sections. A lock is associated with a set of shared data. Only one thread at a time can acquire the lock and is allowed to access to data. As shown in figure 1 the region between the acquisition and release of a lock is called a critical section. Most processors provide a atomic *read-modify-write* primitive to support locks. e.g Intel provides a BTS instruction [2] which can used to atomically test and set a bit. It must be noted that the onus of correctly using locks to restrict shared data access is on the programmer. He has to ensure that all threads acquire the correct lock associated with the data before accessing the data.

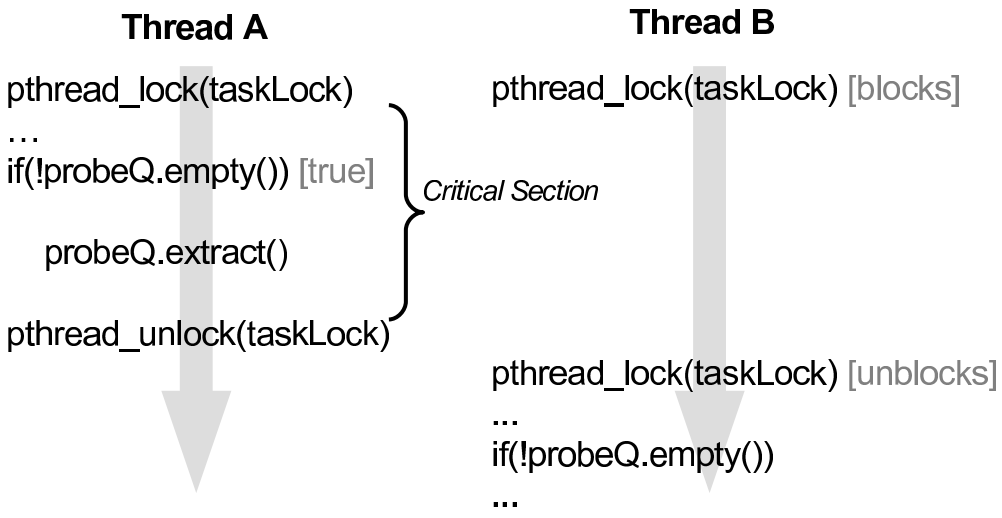


Figure 1: The region between *lock* and *unlock* defines the critical section

A variant of locks used for synchronization is a barrier. As opposed to locks which are used for data access, barriers are largely used as rendezvous point for threads. Threads that arrive at a barrier are not allowed to proceed till the remaining threads arrive at the barrier as well.

Another type of synchronization primitive of significant importance is a conditional variable. Conditional variables can be used by threads to block till (wait for) the condition changes. However, hardware support for conditional variables is extremely limited. Conditional variables are versatile language constructs which can be used to write complex event driven applications, but due to limited hardware support, actual conditional variable usage is minimal and fraught with risk of errors and other inefficiencies.

Transactions have long been used in database systems, however they present a increasingly attractive alternative to conventional synchronization primitives. A transaction [20] comprises of a series of read and write operations that provide the failure- atomicity and serializability. Failure-atomicity states, a transaction must either execute to completion, or in the presence of failures, must appear not to have executed at all. Failure-atomicity provides an all-or-nothing property of execution and guarantees a data structure remains in a consistent state, even in the presence of failures. Serializability is an intuitive and popular consistency criterion for transactions. Serializability requires the result of executions of concurrent transactions to be as if there were some global order in which these transactions had executed serially [20]. Serializability is similar to sequential consistency with regard to memory operations. Lamport [22] defined an execution to be sequentially consistent, if the results of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program. Similarly, an execution of transactions is considered serializable if it appears as if all transactions were executed in some sequential (serial) order with no interleaving within transaction boundaries. While the concept of transactions is simple and convenient for programmers to reason with [12], processors today provide only restricted support for such transactions in their instruction sets. Additional details about support offered by current processors is discussed in section

4

Transactional memory (TM) [6, 13, 17, 27, 32] has been proposed as a way of supporting transactions in hardware. In TM, the programmer simply specifies when a transaction begins and ends, and the system guarantees its atomic execution regardless of how other code in the program is executed. Transactional memory implementation is discussed in section 5.2. Transactions and transactional memory address many of the issues in parallel programming detailed below.

Most architectures have some sort of hardware support for synchronization, largely consisting of single-variable atomic read-modify-write operations that can be used to implement locks. Traditionally, locks have been the synchronization mechanism of choice for programmers and have been extensively used in various software such as operating systems, database servers, and web servers. Critical sections provide an intuitive interface for reasoning about data sharing because they trivially satisfy serializability. Today, critical sections are arguably the most popular abstraction for reasoning about correctness and coordinating sharing in multi-threaded programs.

1.2 Multi-threaded programming considerations

While writing multi-threaded programs requires trade-offs between various characteristics, we consider three important aspects: 1) ease of programming, 2) performance, and 3) energy efficiency of program.

1.2.1 Programmability

Programmability, refers to the ability to write a correct program easily, and is perhaps the determining factor, for widespread adoption of multi-threaded programs [7]. Composability, refers to the fact that atomicity of transactions is guaranteed regardless of how other code is written, and is a major factor in determining programmability of parallel programs. Critical sections once programmed correctly, should execute correctly, irrespective of how other code is written. This helps modularity and is essential for programmability. One of the major attractions of transactions, is its support for composabilty, which most current systems lack. Common cases where lack of composability affect programmability is using

the incorrect lock and forgetting to use a lock. In addition, synchronization waiting on signals also causes problems of deadlock when a particular code in incorrect and lacks the signal.Figure 2 shows this situation.

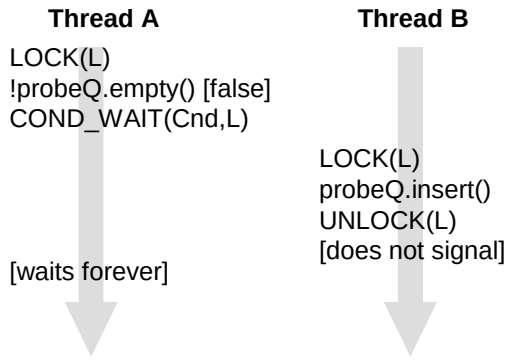


Figure 2: Absence of signal causes deadlock

Apart from correctly coded critical sections working together, the ease of writing critical sections themselves poses a non- trivial problem. Critical sections that hold multiple locks need to acquire them in the same order to prevent deadlocks. Figure 3 represents this situation.

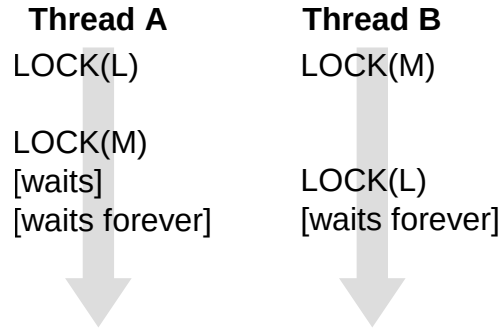


Figure 3: Change in order of locking causes deadlock

Granularity of synchronization is another aspect that affects programmability. When locks are used at a finer granularity, the number of locks needed for synchronizations increase, increasing the chances of error. In addition, the performance may degrade as locking involves a performance overhead. Use of locks at a coarser granularity certainly simplifies programmability, but has performance implications that need to be considered.

1.2.2 Performance

As stated in section 1.2.1, the granularity of synchronization has major performance implications. Depending on the consistency model [5] supported by the system, synchronization operations on most popular computer systems require memory fences or other kinds of safety primitives to ensure sequential memory consistency. Consequently, these operations are implemented as libraries, which the programmer calls to lock or unlocking a particular lock variable,

Memory fences or barriers affect how loads and stores are issued. For example, the Alpha [34] model provides two different fence instructions, the memory barrier (MB) and the write memory barrier (WMB). The MB instruction can be used to maintain program order from any memory operations before the MB to any memory operations after the MB. The WMB instruction provides this guarantee only among write operations. Both barriers degrade performance by preventing masking of memory latencies.

The need to link to external libraries, as well as fences contribute to the overhead of synchronization. As locking forces serialization of access to data, finer locking allows more data to be accessed concurrently, increasing performance. The trade-off between parallelism and performance within each thread arises because execution of lock and unlock primitives may require more time than the execution of a small critical section protected by those primitives. To amortize locking overheads, programmers often combine multiple critical sections into a single larger one, which unnecessarily serializes execution.

1.2.3 Energy Efficiency

With increasing number of multi-core and multi-processor machines, the power budget available for each of the subsystems is limited. In this scenario, energy efficiency is a major consideration in processor architecture. It is also one of the major motivations for this work.

In many parallel applications, a significant amount of energy is expended during synchronization [23]. Spinning at locks,barriers and conditionals are largely responsible for the wasted energy. Spinning refers to the phenomenon where a thread repeatedly reads the lock or conditional variables to check if their value has changed. One the major objectives of

this work is to eliminate energy spent during synchronization.

1.3 Contribution

Transactional memory address many of the issues described above. Transactional memory avoids overheads associated with using locks to protect critical section, eliminates the need to create and manage locks, and prevents non-atomic execution of a correctly written transaction (Figure 6).

Unfortunately, existing transactional memory systems only provide these benefits for atomicity, while the problem of non-composable and error-prone conditional synchronization remains. The main contribution of the work presented in the thesis is *EasySync*. It presents a technique for writing efficient and composable conditional critical section. It provides the benefits of *programmability* and *performance* which transactions provide for lock based critical sections, but more importantly it provides a energy efficient way to implement *all* synchronization.

Additional contributions of this thesis is to evaluate the impact of TM systems augmented by *EasySync* on existing programs, as well as a broader evaluation of the impact of efficient conditional synchronization on considerations while designing parallel algorithms. This work is a logical extension of *EasySync* which essentially completes the support offered by TM systems to synchronization by removing a critical deficiency, and allowing efficient conditional synchronization.

CHAPTER II

EASYSYNC

2.1 Background and Motivation

Parallel programs are hard to write, primarily because their execution must be carefully orchestrated through synchronization, which is very hard to do correctly [7, 28]. Adding unnecessary synchronization leads to lower parallel speedups and possibly deadlocks, while omitting necessary synchronization leads to non-deterministic behavior and may also lead to deadlocks. Despite the key role that synchronization plays in both performance and correctness of parallel programs, existing hardware support for synchronization largely consists of single-variable atomic read-modify-write operations that can be used to implement locks. Efficient implementation of synchronization is especially important for Simultaneous Multi-threading (SMT) [37] processors. This has prompted some recent SMT processors to provide additional support to improve overall processor efficiency. For example, the Intel © Xeon © processor with hyper-threading supports a `PAUSE` instruction which can be used to slow down processor execution while spinning [1]. The canceled Alpha 21464 processor supported a single-cache-block `wait` instruction that stalled the processor until the specified cache block was modified by other processors [10]. This instruction could be used to efficiently implement wait-signal synchronization thorough conditional (flag) variables. However, it did not support waiting on arbitrary conditions that involved multiple variables.

If used correctly, locks can provide atomic execution of larger critical sections, whereas conditional variables can enable one thread to wait until a condition is satisfied by other threads. Locks and conditional variables are also often used together when the condition check and the action that should follow it must be performed atomically, like in Figure 4 where the thread needs to get an element from either `probeQ` or `taskQ`, and must wait for one of the queues to get an element if both queues are currently empty. Note that the `pthread_cond_wait` primitive releases the `taskLock`, waits for a signal on the `qNotEmpty`

conditional variable, then re-acquires the `taskLock`.

```
...
ptread_lock(taskLock);
  while(taskQ.empty()&&probeQ.empty())
    pthread_cond_wait(qNotEmpty,taskLock);
  if(!probeQ.empty())
    t = probeQ.extract();
  else
    t = taskQ.extract();
pthread_unlock(taskLock);
...
```

Figure 4: Example code with locks and conditional variables.

This code may seem simple, but is actually subject to a very complex set of trade-offs and modularity issues. One such issue is that `taskLock` provides atomic execution for this code only if the same lock is used for all other code that modifies `probeQ` or `taskQ`. For example, Figure 5 shows how Thread A's code (which is written correctly) can execute non-atomically because Thread B's code uses probeQ without first acquiring `taskLock`. Debugging is further complicated because Thread B's incorrect code *does* execute atomically in this example. Overall, atomic execution of a lock-based critical section depends on how the rest of the code is written, making lock-based synchronization very difficult to program and debug.

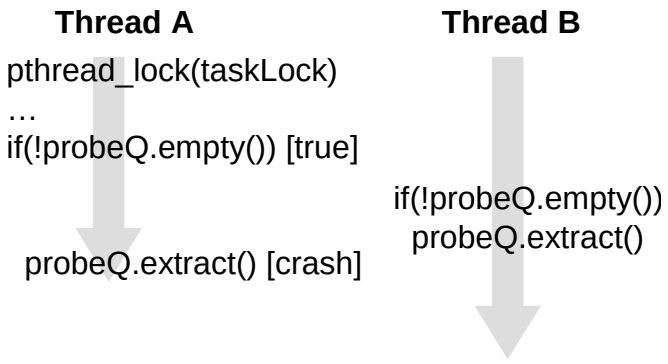


Figure 5: A critical section can execute non-atomically.

Code based on conditional variables has a similar problem. For example, correct behavior of the wait in Figure 4, depends on all other code correctly signaling on `qNotEmpty`

whenever the underlying condition changes, i.e. when an element is inserted into one of the queues. If some other code fails to signal, or signals on a different conditional variable, our example code (which is written correctly) can wait forever.

To ensure atomicity of a particular set of variables, all their writes and most reads have to be protected by the same lock. Similarly, a conditional variable is associated for a specific actual condition, and any change of the underlying condition has to be signaled using that conditional variable. This introduces additional sources of program errors, such as using a wrong synchronization variable or forgetting to use one. Such synchronization is also not *composable*, because correctness of synchronization in one fragment of code depends on how all other code in the program is synchronized.

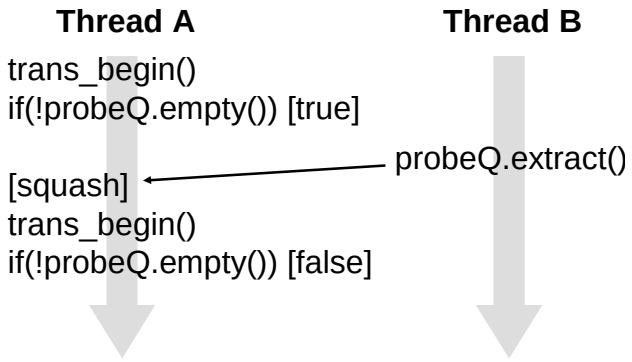


Figure 6: Atomicity of a transaction is guaranteed by the system.

As discussed in the introduction, transactional memory(TM) is a hardware mechanism to support transactions. To implement transactional memory, the system tracks the variables the transaction reads and monitors writes to these variables by other threads. If such a write is observed, the system squashes the transaction, undoing all its side effects, and restarts it. Transactional memory avoids overheads associated with using locks to protect critical section, eliminates the need to create and manage locks, and prevents non-atomic execution of a correctly written transaction (Figure 6).

Unfortunately, existing transactional memory systems only provide these benefits for atomicity, while the problem of non-composable and error-prone conditional synchronization remains. A study [7] indicated that bugs that lead to deadlocks on conditional variables

occur in as many programs as do lock-related data race problems. Consequently, in order to fully leverage the potential of transactional memory, efficient and composable support for conditional synchronization without conditional variables is essential.

```
...
trans_begin();
  while(taskQ.empty()&&probeQ.empty()){
    trans_end();
    trans_begin();
  }
  if(!probeQ.empty())
    t = probeQ.extract();
  else
    t = taskQ.extract();
trans_end();
...
```

Figure 7: Conditional synchronization using spin-waiting in transactional memory.

Conditional synchronization through spin-waiting is composable, but suffers from lack of efficiency. An example of such code is shown in Figure 7, where the thread repeatedly checks the condition it is waiting for. This code uses no condition variables and is composable: the wait will end when there is an element in one of the queues, and the thread that inserts that element does not need to signal this condition to the waiting thread. However, spin-waiting is inefficient: it expends energy on useless activity and the resource contention by waiting threads may slow down progress of other threads that are doing useful work. Spin-waiting code also expresses the synchronization intent of the code less directly, which may be a problem in porting, maintaining, and debugging.

Our EasySync mechanism supports composable conditional synchronization in transactional memory systems. EasySync extends transactional memory with a *transaction wait primitive*, which is used to inform the system that the program is waiting for a condition to be satisfied. The hardware can then leverage existing transactional memory mechanisms to stop the thread's execution until one of the values used in the evaluation of the condition changes. Just like transactional memory eliminates lock variables, EasySync eliminates the need to create, manage, and use conditional variables. Also, the wait in EasySync ends

successfully when its condition is met, regardless of how other threads are synchronized; this parallels transactional memory's guarantee of atomic execution regardless of how other threads use (or do not use) synchronization.

In essence, EasySync completes transactional memory's synchronization support to allow, for the first time, efficient and composable conditional, atomic, and conditional-atomic execution. Whereas existing hardware support for transactional memory allows efficient execution of composable critical sections, EasySync-enabled hardware allows efficient execution of composable conditional critical regions (CCRs) [18].

```
...
synchronized when{
    case(!probeQ.empty()) :
                        t = probeQ.extract();
                        break;
    case(!taskQ.empty())  :
                        t = taskQ.extract();
                        break;
}
...
```

Figure 8: Example of synchronization code from Figure 4, written using advanced CSP-inspired synchronization.

As result, EasySync can efficiently and directly support more advanced language constructs to express conditional synchronization in shared-memory programs. Figure 8 shows our example code using constructs inspired by CSP [19] and guarded statements [9], where the `when` statement is similar to a C/C++/Java `switch` statement that waits for one of its conditions to become satisfied. Figure 9 shows a straightforward transformation of this code to use EasySync.

Although it directly supports advanced composable synchronization constructs, EasySync also easily implements legacy code based on wait-signal synchronization, which can be automatically converted to use EasySync's efficient waiting (Figure 10).

```
...
trans_begin();
  if(!probeQ.empty())
    t = probeQ.extract();
  else if(!taskQ.empty())
    t = taskQ.extract();
  else
    trans_wait();
trans_end();
...
```

Figure 9: Code from Figure 8 transformed for EasySync.

```
...
trans_begin();
  if(taskQ.empty()&&probeQ.empty())
    trans_wait();
  if(!probeQ.empty())
    t = probeQ.extract();
  else
    t = taskQ.extract();
trans_end();
...
```

Figure 10: Example code from Figure 4 converted to EasySync.

2.2 Implementation of EasySync

2.2.1 ISA extension: the `XWAIT` instruction

Hardware support for transactional memory already has instructions to begin and end a transaction (e.g. `XBEGIN` and `XEND` from [6]). In our C-language examples, `trans_begin()` and `trans_end()` directly translate into these instructions. To implement EasySync, we add a transaction wait instruction (`XWAIT`), and in our examples `trans_wait()` directly translates into this instruction. When the processor executes a `XWAIT` instruction, it knows that the synchronization condition checked by the current transaction is not satisfied and that the transaction must be aborted and restarted when the condition is satisfied or might be satisfied.

We note that there are many correct implementations of the `XWAIT` instruction. The simplest implementation would be to squash the transaction and restart it immediately

(Figure 11). However, this implementation is no better than spin-waiting in terms of energy and performance. The advantage of EasySync becomes apparent when we consider more advanced implementations.

2.2.2 Efficient Implementation of `XWAIT`

An ideal implementation of the `XWAIT` instruction postpones re-execution of the transaction until the exact time when it can successfully complete without reaching a `XWAIT` instruction again. Because a transaction can execute arbitrary code before reaching either `XWAIT` or `XEND`, it is very difficult to determine exactly when this condition is satisfied. Fortunately, we know that the transaction will re-execute in exactly the the same way if it reads exactly the same values. Therefore, when a transaction reaches the `XWAIT` instruction, we monitor accesses to memory locations accessed by the waiting transaction, and trigger a squash and re-execution when we observe a write (by another processor or device in the system) to one of these locations.

We note that existing transactional memory implementations already detect writes to locations read by an uncommitted transaction. This detection is needed to find atomicity violations and trigger a squash and restart of one of the transactions. In TCC [13], for example, the processor marks cache blocks that have been read and written by the currently executing transaction. When a transaction completes, it commits by grabbing the bus and writing back its modified blocks. Every other cache observes each of these write-backs, and squashes its local transaction if the block is in the cache and marked as read by the local transaction. To implement EasySync in TCC, we only need to stop the transaction when it executes the `XWAIT` instruction, and the regular TCC mechanisms squash and restart that transaction when there are changes to its input variables (Figure 12). Actually, TCC checks entire blocks instead of words or bytes, so a waiting transaction might be squashed because a committing transaction has modified a variable that just happens to be in the same cache block as one of the variables read by the waiting transaction. If such false squashes are frequent, EasySync still provides correct execution but the unnecessary activity reduces its energy and contention benefits. However, we note that this loss of efficiency is

not intrinsic to EasySync – it is an artifact of a block-granularity transactional memory implementation, where frequent false sharing creates numerous unnecessary squashes and performance problems with or without EasySync.

Unbounded Transactional Memory (UTM) [6] and Virtualized Transactional Memory (VTM) [32] allow transaction data to spill from caches into local main memory, and detect transaction conflicts when one overwrites the value read by another (as opposed to when the overwriting transaction commits in TCC [13]). However, as in TCC, a conflict in UTM and VTM still results in squashing the reading transaction, and our EasySync `XWAIT` instruction can be implemented in the same way we describe for TCC: stop the transaction until the underlying TM protocol squashes it.

LogTM [27] is an interesting case because its hardware detects conflicts and can then uses a conflict resolution handler to decide which transaction(s) to squash. To implement EasySync correctly on top of LogTM, the conflict resolution handler must be changed to check if one of the transactions involved in a conflict is waiting (has executed `XWAIT` and has not yet been squashed). If a waiting transaction is involved, the conflict should always be resolved by squashing the waiting transaction and allowing the writer to proceed. We note that a conflict between two waiting transactions can not happen, because once the wait begins the transaction can not issue writes (or any other instructions) until it is squashed.

Another possible complication for EasySync implementation would be a TM system that keeps track of the "oldest" transaction and always resolves conflicts in favor of this "oldest" transaction to guarantee forward progress. A transaction that might still execute the *XWAIT* instruction should not be given the "oldest" status in such a system. In our current implementation, the system does not know which transactions might execute `XWAIT` until they actually do so or reach `XEND`. As a result, no transaction can be granted the "oldest" status until it ends, defeating the purpose of the "oldest" designation. For these systems, `XBEGIN` instruction could indicate whether the transaction is completely free of `XWAIT`. Also, an additional `XNOWAIT` could be used to indicate the point past which the transaction can no longer reach an `XWAIT`. We believe that simple reach-ability analysis can be used by the compiler to insert these instructions automatically. However, such an

implementation is out of the scope for this work because recent hardware TM proposals do not need it.

2.3 Using Legacy Code with EasySync

For a new synchronization scheme to be feasible, it should require little or no change to existing (legacy) code. For this reason, we implemented and tested a library with existing synchronization primitives using transactions and our new `XWAIT` instruction. With this library, existing code can run unchanged if it uses dynamically linked libraries. Statically linked applications must be re-linked.

Our library replaces `lock()` and `unlock()` calls with transaction begin and end primitives, replaces `cond_wait()` calls with our new transaction wait primitive, and eliminates calls to `cond_signal()` because they are unnecessary. We replace the `barrier()` function with an EasySync-based one. Figure 13 shows this `barrier()` function, with some declarations omitted for clarity.

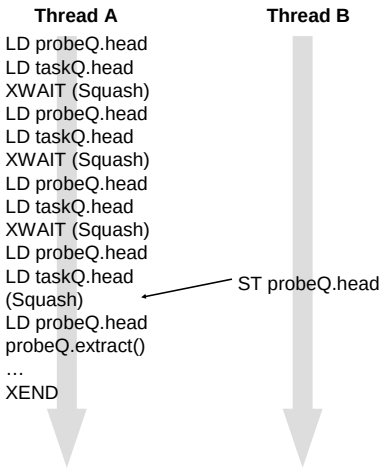


Figure 11: A naive implementation of our `XWAIT` instruction

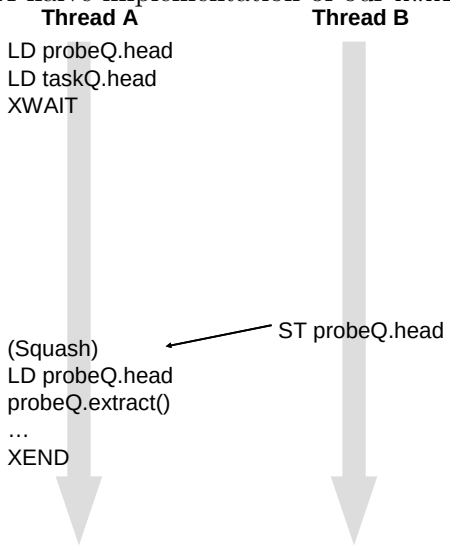


Figure 12: Efficient implementation of our `XWAIT` instruction

```
void barrier(bar_t *bar, int nthr){
  lsense=!bar->gsense;
  trans_begin();
  if(bar->cnt==nthr-1){
    bar->cnt=0;
    gsense=lsense;
  }else
    bar->cnt++;
    trans_end();
    trans_begin();
    if(gsense!=lsense)
      trans_wait();
    trans_end();
  }
  trans_end();
}
```

Figure 13: Barrier implemented using TM and EasySync

CHAPTER III

SOFTWARE IMPACT

3.1 Impact on Programmability

EasySync provides a powerful tool to programmers for writing simple yet energy efficient parallel programs. While it provides significant advantages to existing programs, in order to better leverage the potential of the system, we also modified a few Splash-2 applications, namely Barnes,Radiosity, and Raytrace. The objective was two fold, first to expose more spin-wait activity that was difficult to identify using automatic compiler level techniques, and secondly, improve readability of code, by replacing complicated synchronizations with simpler EasySync primitives. This set of applications is intended to represent future applications which are written with transactional memory and conditional synchronization in mind. For EasySync, these applications expose more waiting activity because we replaced ordinary do-while waiting loops with EasySync primitives. A broader impact at the algorithmic level is discussed in section 3.2.

Figure 14 shows an simplified example of a modification to Barnes while, figure 15 shows a simplified version of one of the more elaborate modifications in Radiosity. Raytrace has wide-ranging modifications mainly to improve readability. Full code listing of the modified files are provided in the appendix. The original source code for the applications can be obtained from [4].

In Barnes, the while loop spinning on the *done* variable was eliminated by replacing it with a conditional critical section. This shows a simple and direct application of the `trans_wait` primitive. In Raytrace, some of the synchronization code was completely rewritten for better readability. Radiosity was one of the more interesting applications modified. Each thread in the Radiosity has its own task queue. However, it has a highly unstructured nature, which may cause significant load imbalances. The *task stealing* technique is used to address some of the load imbalance issues.Task stealing is a technique in which a processor

```
...
while(!Done(r)) {
  /* wait */
}
...
```

(a) Original Code

```
...
trans_begin();
  if (!Done(r))
    trans_wait();
trans_end();
...
```

(b) Modified Code

Figure 14: Modifications to Barnes

which has finished processing all tasks in its queue, tries to *steal* a task queued on a different processor. The processor pool for task stealing have to be topologically close enough, so as to have a short data transfer time as compared to the time required to execute an average task. On a system implementing task stealing, when a thread reaches a barrier, it has to check two conditions; first, it checks for the exit condition, i.e. if all other threads have reached the barrier then the solution has converged and the function exits, secondly, if while waiting at a barrier, a running thread has created additional tasks which could be stolen, then the thread at the barrier has to exit the barrier and process the task. Lacking arbitrary conditional synchronization, the approach taken in the code is as follows. When a thread has no tasks to perform, it enters a spin-wait loop where it spins for a fixed amount of time checking if all other threads have finished processing tasks. If all other tasks have not finished, the thread then does a search to check if it can steal a task, failing which it enters the barrier spin loop again. Figure 16 gives a overview of this process. The code has already been depicted in figure 15

All three applications demonstrate both qualitative and quantitative advantages with baseline transactional memory and with EasySync. In terms of qualitative advantages, code readability is increased and code size is reduced for all three applications, with significant improvement in Radiosity. The quantitative gains in these applications also are evaluated and described in Section 6.2.

```
...
while(global->pbar_count < n_processors) {
  /* spin wait on barrier counter for a while
     exit loop if all processes enter barrier */
  ...
  t = DEQUEUE_TASK(...);
  /* if dequeue task successful, decrement
     barrier counter and restart process */
  ...
}
...
```

(a) Original Code

```
...
trans_begin();
if (global->pbar_count < n_processor) {
  /* read task queue */
  if (tq->top) {
    /* get task from queue, decrement barrier
       counter and restart process */
    ...
  }
  else
    trans_wait();
}
trans_end();
...
```

(b) Modified Code

Figure 15: Modifications to Radiosity

3.2 Algorithmic Impact

For the first time, Transactional Memory augmented with EasySync provides the programmer with a complete set of tools to easily program multi-threaded applications. Even though existing programs perform well with EasySync coupled with compiler level techniques and manual code modifications exposing additional synchronizations, we believe that that the proposed system has a wider impact at the algorithmic level, especially since a different set of assumptions and trade-offs have to be considered.

While designing parallel programs is a complex process without a standardized method, a generalized approach commonly used involves four distinct steps [11], namely partitioning, communication, agglomeration, and mapping. The first two steps explore various algorithmic options focusing on concurrency and scalability. In the third and fourth steps attention shifts to locality and other performance related issues. The four steps are illustrated in figure 17 and can be summarized as follows:

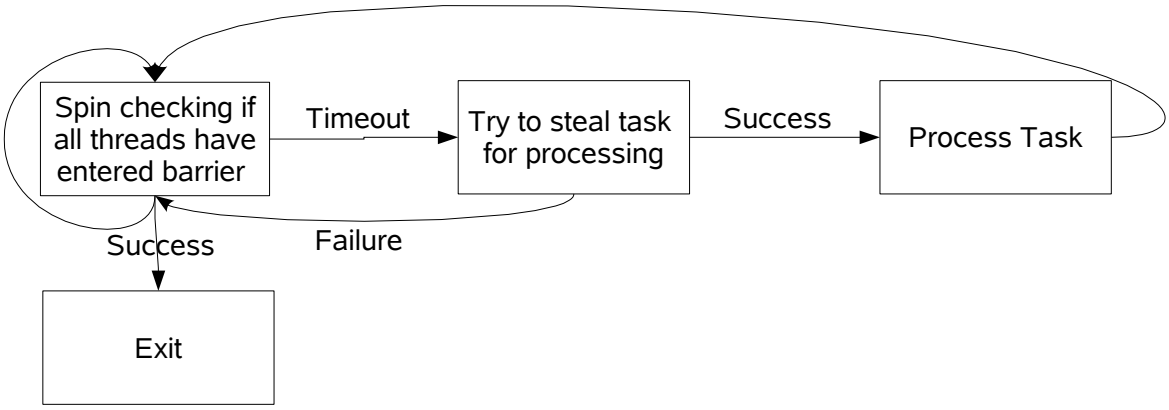


Figure 16: Synchronization involving task stealing in Radiosity.

1. *Partitioning.* At this stage, parallelism is the main focus. The computation as well as the data are analyzed and decomposed into smaller potentially concurrent tasks. The underlying hardware architecture is not taken into consideration and factors such as data transfer time, etc. are also not considered.

2. *Communication.* The communication required to co-ordinate tasks execution is determined , and appropriate communication structures and algorithms are defined.

3. *Agglomeration.* It is at this stage that the system hardware is taken into consideration. The partitioning of tasks and the communication required to co-ordinate them are usually at odds with each other, with a finer decomposition requiring high communication. It is at this stage the underlying topology of the system is considered along with communication costs and if necessary tasks are combined into larger tasks to improve overall performance or to simplify programming.

4. *Mapping.*Each task is assigned to a processor to maximize utilization and reduce overall execution time.Mapping is a complicated process depending on the algorithm with static or dynamic load balancing techniques used to maximize utilization.

While writing programs with Transactional Memory in mind, the biggest difference is that errors in task decomposition do not necessarily affect the outcome of the program. Two

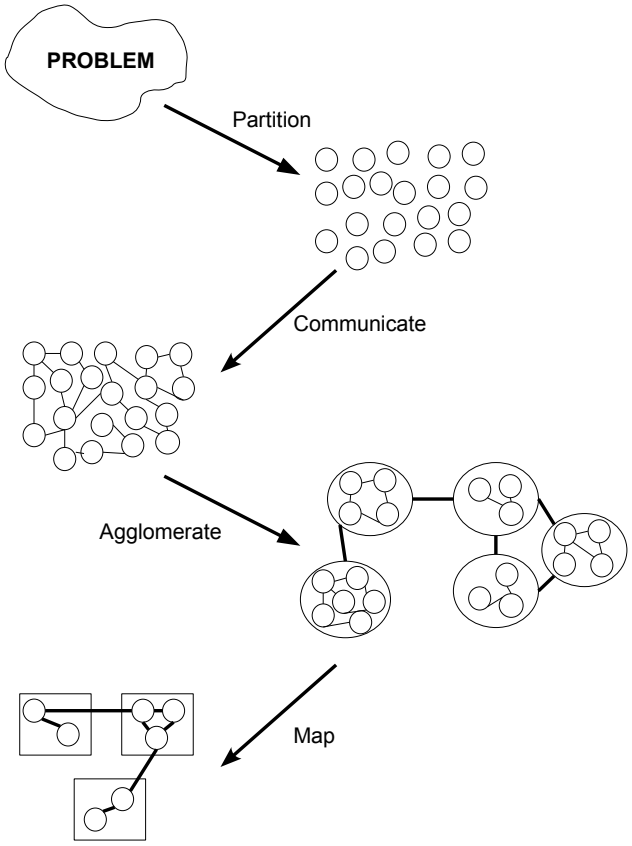


Figure 17: Steps involved in designing a parallel program

tasks incorrectly identified as concurrent and executing in parallel will not crash the program, TM system handles such a situation by squashing one of the tasks and re-executing it, thereby automatically enforcing serialization. This fact can be exploited to concurrently execute tasks which rarely conflict. Transactional memory effectively exposes more parallelism as tasks not provably parallel can be executed concurrently so long as they do not *actually* conflict.

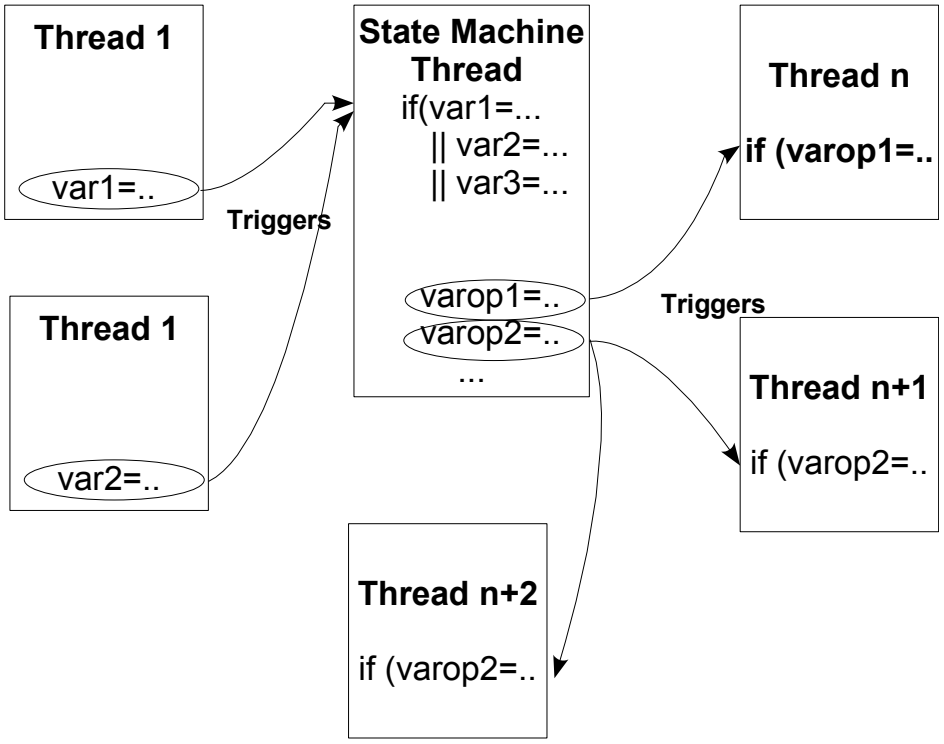
TM does not significantly affect the communication aspect of parallel programs at the algorithmic level. However, it does make the actual programming of communication constructs much simpler.

It is at the agglomeration and mapping stages that the potential of TM in general and EasySync in particular are noticeable. Smaller concurrent tasks are combined many a times to 1) simplify programming and 2) reduce synchronization and communication

overhead. Since TM essentially eliminates locks, it considerably simplifies programming. Transactions provide a versatile alternative to locks while not necessarily adding to program overhead, this in turn allows the programmer to use synchronization at a finer level, and obviates the need to unnecessarily serialize code, reducing concurrency, merely to save on synchronization.

The major challenge while mapping a parallel program on to the underlying hardware is *load balancing*. The ability to efficiently synchronize on arbitrary conditions provided by EasySync is invaluable for load balancing. The code impact of EasySync on load-balancing has already been discussed in section 3. At an algorithmic level, conditional synchronization provides an ability to implement **true event driven** behavior. This, we believe, is the biggest impact of EasySync on parallel programming. EasySync provides programmers the ability to generate arbitrary events and invoke threads based on the events. Traditionally,event based programming has been mainly been used in graphic user interface (GUI) context. For example, in the Microsoft Windows operating system events are generated by operating system, usually in response to user interaction or timer related events and processed a main `while` event loop in the program [3]. These events are generated by the operating system at a higher level by processing various lower-level events and other inputs. True events are generated at a very low level in the form of interrupts, which are processed by interrupt handlers located at predefined addresses. On an interrupt, the the processor automatically jumps to these predefined addresses. EasySync provides the same ability directly to the programmer without the need of having to go to several layers of operating systems, thereby improving performance and energy efficiency. Besides, EasySync does not require the programmer to make complicated application programming interface (API) calls to define and wake-up on events, it just needs the `trans_wait` call and a few if statements to parse out the event. EasySync makes it simple and efficient for the programmer to use events and implement logic using structured design or transaction analysis methodologies. Event handlers, at a more abstract level are state machines [21]. Figure 18 illustrates an example usage of EasySync for state machine based programming. The `state machine` thread is idle till a write occurs to one of the variables on which it

is waiting. This then triggers the thread which performs some computation and in-turn may write variable on which other threads are waiting. It must be noted that the code shown ignores various other considerations for the sake of simplicity. For example, if the main transaction in `state machine` thread has not completed by the time a conditional variable is re-triggered (rewritten), the transaction will squash. This may require the `state machine` to be programmed intelligently, by splitting into multiple threads, queuing and other techniques, normally used in state machine design.



(a) State machine visualization

```
void state_machine(){
 while(1){
  TRANS_BEGIN;
  if((var1==cond1) || (var2==cond2)...){
    if (var1==cond1){varop1=...}
    if (var2==cond2){varop2=...}
    ...
  else
    TRANS_WAIT;
  }
  TRANS_END;
 }
}
```

(b) Code for state machine thread

Figure 18: Equivalence of state machine and conditionally triggered thread

CHAPTER IV

RELATED WORK

Lock speculation and elision mechanisms [26, 30, 31] use transactional execution to execute lock-based code with more concurrency and with less locking overheads. Speculative synchronization [26] also allows speculation through other library synchronization primitives, such as barriers and flags. However, this work is aimed at reducing synchronization overheads and increasing concurrency when executing existing code based on locks, barriers, and flags. The focus of our work is to provide support for efficient and composable conditional synchronization for transaction-oriented code.

The Thrifty Barrier [24] saves energy by slowing down and stopping threads that wait inside barrier synchronization. To avoid unnecessary checks of the barrier variable, the Thrifty Barrier waits for coherence invalidations caused by writes to the barrier variable. The support used for Thrifty Barrier is only suitable for implementation of barriers, conditional variables, and other synchronization primitives that only need to wait on a single variable, whereas EasySync provides a more general support that allows efficient waiting for arbitrary conditions without forcing the programmer to create and manage special conditional variables. As a result, Thrifty Barrier only alleviates some energy-related problems when using existing synchronization primitives, while EasySync provides efficient support for more programmer-friendly composable conditional synchronization integrated into transactional memory systems.

The `PAUSE` instruction in recent Xeon and Pentium 4 processors from Intel can be used to introduce a delay in the spin-waiting loop, reducing the energy consumption and resource contention by the spinning thread. In contrast, the EasySync `XWAIT` allows the processor to stop completely until there is a change in the condition being checked. EasySync also allows efficient and straightforward implementation of conditional critical sections and other advanced conditional synchronization, whereas the `PAUSE` only adds more coding complexity

to spin-looping code.

Related research work on composable synchronization includes Hoare's conditional critical regions (CCRs) [18], in which a boolean condition can guard a critical section. Variants of CCRs are present in several programming languages and their extensions [14, 15, 25, 35], and include a recent implementation of CCRs using a software transactional memory system [16]. A drawback of all these CCR approaches is the high overhead and a complex runtime environment needed to correctly execute CCRs on existing hardware. EasySync allows, for the first time, efficient execution of composable CCRs and other advanced synchronization constructs.

Support for transactional memory [6, 13, 17, 27, 32] efficiently provides composable critical sections. Our EasySync mechanism extends and complements transactional memory by leveraging existing transactional memory mechanisms to also provide composable conditional waiting and conditional critical regions. In essence, EasySync does for conditional variables and conditional synchronization what transactional memory did for locks and critical sections: it frees the programmer from having to use specific variables to express synchronization conditions and allows correct condition-waiting to succeed when the real condition is satisfied without the need to signal this condition using a special variable.

CHAPTER V

EVALUATION SETUP

5.1 Simulation Environment

We conduct detailed execution driven simulations by extending the SESC [33] cycle-accurate execution-driven simulator. A chip-multiprocessor architecture with private L1 caches, a shared Level 2 cache and transactional memory support is used as a baseline for our simulations. We vary the number of simulated cores from 2 to 16. . Table 1 shows the major characteristics of the architecture. The overall architecture of the system is shown in figure 19.

Table 1: Architecture Characteristics used for simulations

Processor	
Freq., Issue	5-GHz, 4-wide out-of-order
Execution Units	2 Alu, 2 Mul, 2 Div, 3 Fp, 2 Branch
Branch Predictor	Hybrid 16 kb meta, 16kb local
Branch Penalty	17 cycles
Load/Store	2 Load, 2 Store , all 56 entry
Reorder Buffer	176 Entries
Memory	
L1 Cache	32 kB, 32B lines,4-way
L2 Cache	1 MB, 1024B lines, 8-way
L1 Ports,Hit/Miss delay	2 ports, 2 /2 cycles
L2 Ports,Hit/Miss delay	(1+ 1snoop) ports , 9/11 cycles
Memory bus	10 GB/s, Split transaction bus

5.2 Transactional Memory Implementation

Our implementation of transactional memory employs a speculative L1 cache which buffers all writes and prevents write-backs to L2 until the transaction commits. Similarly, when a transaction reads an L1 cache block, the block is marked as having been read and any external writes matched to these blocks cause a squash and re-execution of the transaction. The most current version of data is provided to the transaction. Transaction commits are

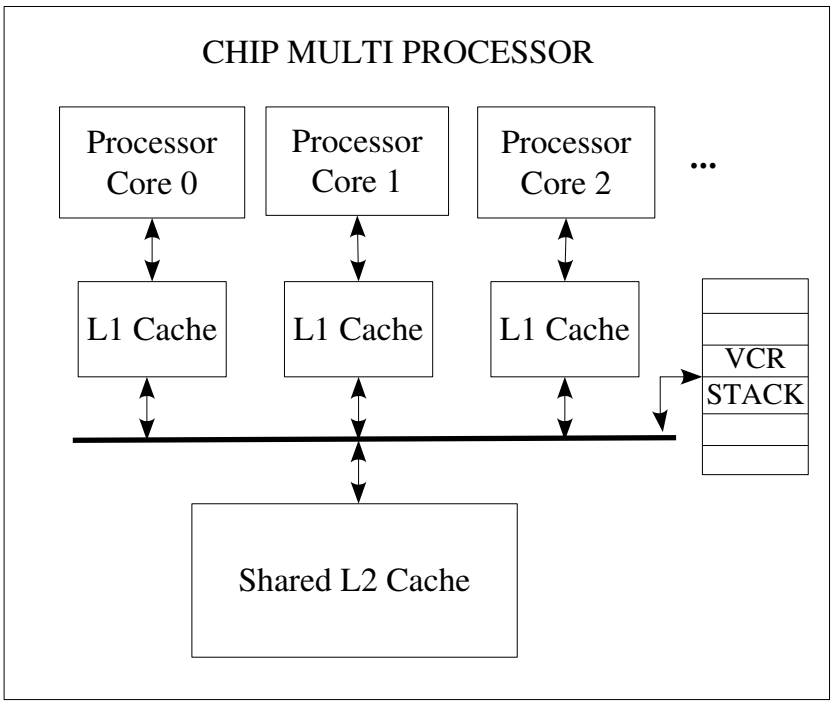


Figure 19: Simulated architecture

performed in a lazy manner, i.e. when a transaction commits it does not write back all its lines from cache to the L2; instead, these lines are written back only when they are evicted due to lack of space. Squashes cause all lines buffered by the transaction in the L1 cache to be invalidated. Lines marked as read are also invalidated on a squash. We assume an idealized overflow support because such support is orthogonal to our EasySync scheme: none of our waiting transactions overflow the L1 cache, so any overflows affect both the baseline transactional memory system and our EasySync-enabled system equally.

The transactional memory implementation that has be simulated has been implemented for a Chip Multi-processor (CMP) and uses schemes suggested in [29] and [8]. A transaction is speculative when it may perform or may have performed operations that violate atomicity constraints of other transactions. When a transaction is the oldest speculative transaction t becomes non-speculative. When a non-speculative transaction finishes execution, it is ready to commit. The role of commit is to inform the rest of the system that the data generated by the transaction are now part of the safe, non-speculative program state. Among other operations, committing always involves passing the non-speculative status to a successor

transaction. This is because we need to maintain correct sequential semantics in the parallel execution, which requires that transactions in a thread commit in program order. If a transaction reaches its end and is still speculative, it cannot commit until it acquires non-speculative status.

Memory accesses issued by a speculative transaction must be handled carefully. Stores generate speculative versions of data that cannot be merged with the non-speculative state of the program. Only when the transaction becomes non-speculative can its versions be allowed to merge with the non-speculative program state. Loads issued by a speculative transaction try to find the requested data in the local speculative buffer. If they miss, they fetch the closest predecessor version from the speculative buffers of other transactions. If no such version exists, they fetch the data from memory. As transactions execute in parallel, the system must identify any violations of cross-transaction data dependencies. The Epoch IDs are used for this purpose. A data dependence violation is flagged when a transaction modifies a version of a data that may have been loaded earlier by a transaction from another thread. At this point, the consumer transaction is squashed and all the data versions that it has produced are discarded. Then, the transaction is re-executed. To prevent needless squashes caused by false data dependencies system keeps track of accesses an a per-word basis as opposed to a per-line basis which cannot disambiguate accesses to different words in the same memory line. The `Read Exposed (RE)` flag keeps track of the reads. If a word is read without being first written to then this flag is set and indicates a read dependence. A write to a word sets the `Write Modified (WM)` flag. Since a write to a word can potentially cause a conflicting transaction to abort, all writes need to be be broadcast on the shared bus. This causes a significant increase in bus traffic, reducing performance. In order to filter out some of the bus traffic, the `Others Exposed (OE)` flag is used. It is used to keep track of whether a word has been read from the line by some other processor. If the flag is set then a write to that word can cause a abort and needs to be broadcast on the bus. Figure 20 shows the modified cache line structure.

The need for a transaction to read the most recent version of data creates additional complications which need to be handled. Each transaction creates its own copy of accessed

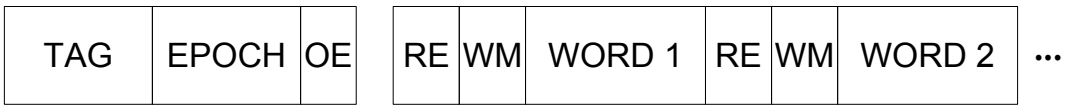


Figure 20: Organization of a Cache Block

line. As different transactions may have written to different words, all versions of the same line need to be combined before forwarding the data to complete a read request. A `version combining register (VCR)` is used for this purpose. When a read request reaches the shared bus, it allocates a VCR from the VCR stack. All caches with a valid line matching the tag, forward the line to the VCR, which then combines them and forwards the resulting line to the requesting cache. This process is illustrated in figure 21.

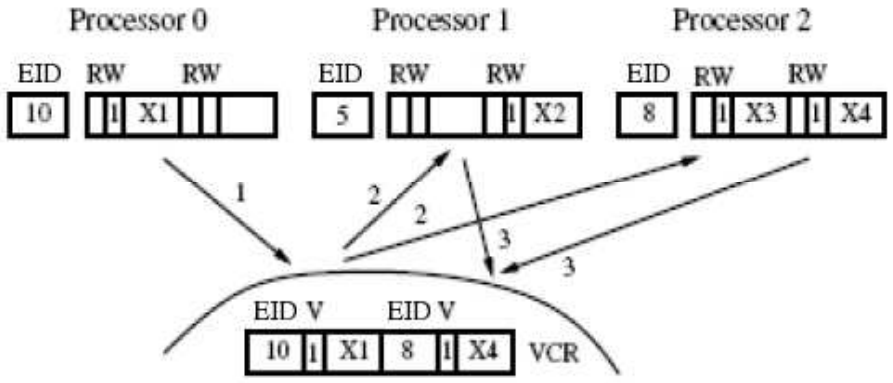


Figure 21: The eviction of a line from processor 0 s cache (left) triggers a transaction in three steps (1-3) that fills a VCR as shown. EID, R, W, and V stand for epoch ID, read exposed, write modified, and valid, respectively. A word is dirty if its W is 1.

5.3 Applications

We evaluate EasySync using the Splash-2 [38] suite of parallel applications. Table 2 lists the applications we simulate and the corresponding inputs. The problem sizes have been chosen such that the applications run a reasonable amount of time in the parallel region, allowing us to clearly observe the effects of threads spinning on locks and other synchronization primitives. Considering this, we also increase the number of time steps for n-body problems, in order to extend the execution and reach a steady state for synchronization. This is

compliant with the nature of these applications in a more realistic setting [36]. Some of the data points in the evaluation are missing due to the inability of our simulator, primarily due to memory constraints, to execute certain applications with larger number of cores. We are investigating this issue, but do not expect significant variations in the trend already demonstrated by these applications.

Table 2: SPLASH 2 Applications and corresponding input sets

Applications	Problem Size
Barnes	2k particles, 8 time steps
Cholesky	tk23
FFT	1M points
FMM	2048
LU	512x512 matrix
Ocean	130x130
Radiosity	-test
Radix	256K keys
Raytrace	Teapot
Volrend	head
water-n2	216 molecules 12 time steps
water-sp	216 molecules 12 time steps

Two distinct sets of applications were evaluated. The first set consists of all applications in the Splash-2 suite, without any source code modifications. To add EasySync support to these applications, we implemented a library of synchronization primitives (locks, flags/-conditionals, and barriers) which employ transactions and the EasySync condition-wait primitive. Subsequently, all the applications were re-linked with the new libraries. This allowed us to evaluate the impact of EasySync on legacy code not optimized for TM. The results presented are for the aggressive library implementation which replaces lock-unlock pairs with transaction begin and end instructions. The EasySync configuration we simulate uses the stop-wait implementation of our new `XWAIT` primitive.

Since all applications in the first set were treated as legacy code, the savings are limited only to synchronization which used library primitives. This reduces the benefits of our scheme because several Splash-2 applications use spin-loops to avoid overheads and complexity of locking and conditionals. These spin-loops do not use any library functions and are difficult to identify as synchronization.

In order to better leverage the potential of the proposed system, we also show results for a few Splash-2 applications, namely Barnes, Radiosity, and Raytrace, in which we rewrote synchronization with transactional memory and EasySync in mind. This second set of applications is intended to represent future applications written for transactional memory. For EasySync, these applications expose more waiting activity because we replaced ordinary do-while waiting loops with EasySync operations.

CHAPTER VI

EVALUATION OF EASYSYNC

While evaluating EasySync, we simulate both the legacy and the optimized application sets. The main focus of the evaluation is the number of spinning instructions (instructions fetched and executed while spinning on a conditionals or barriers) that have been eliminated. While we have not simulated an explicit energy model, we believe that the number of eliminated instructions is representative of the energy savings. This is especially the case because we are comparing a baseline transactional memory system with an EasySync-enabled system, which has nearly identical hardware but executes fewer instructions. In addition to the number of eliminated spinning instructions, we also evaluate the performance impact of EasySync.

6.1 Applying EasySync to Legacy Code

6.1.1 Overall Results

While evaluating the legacy code we simulated the entire Splash-2 suite, with and without EasySync, for an increasing number of processor cores. Figure 22 shows the number of spin-instructions eliminated, normalized by to the total instruction count in the non-EasySync execution. Water-sp gains the most with a maximum of 24% instructions eliminated with 8 processors. The mean across the suite has a maximum value of 8.6%, which is reached at 16 processors. Figure 23 shows the number of cycles spent waiting on synchronization primitives, normalized to the overall execution time. Water-sp spends the most cycles waiting, with a maximum of 56% (reached at 16 processors). The mean number of cycles spent waiting across all the applications reaches a maximum at 16 processors, when this mean is 11.3%. These idle cycles *do not* represent an overhead; instead they represent the time that the baseline processor would have spent fetching and executing *useless* instructions while spin-waiting.

Figure 24 shows the speedup achieved by EasySync. The maximum overhead of EasySync is about .4% which is well within the error range of the simulation. The maximum speedup achieved is 13% for Water-n2, with a mean speedup across all applications being 3.6% max. These results are further analyzed in the next section.

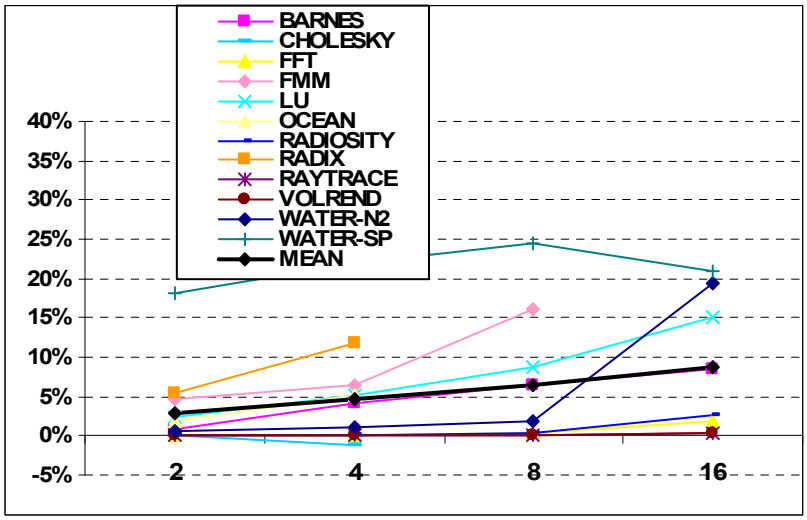


Figure 22: Normalized Spin-Instruction count for various CMP sizes

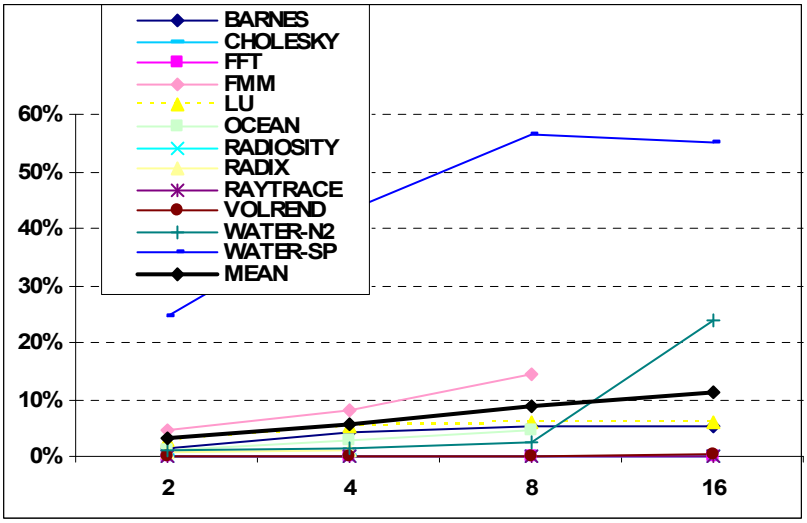


Figure 23: Normalized Idle-Cycle count for various CMP sizes

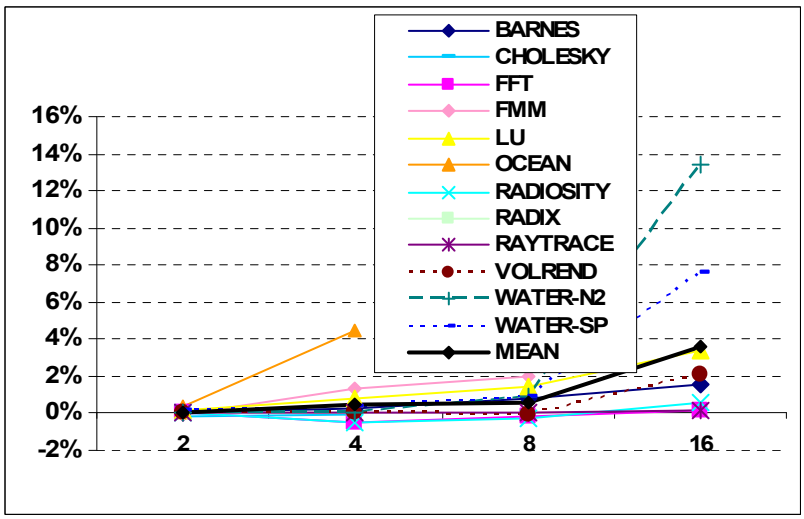


Figure 24: Normalized speedup for various CMP sizes

6.1.2 Analysis

Spinning in barriers is usually caused by load imbalances in the program. On the other hand, spinning on conditionals is generally caused by producer-consumer relationships among threads, when the consumer arrives to the synchronization before the producer does. In general, our results confirm the expected result that, with increasing numbers of processor cores, the number of instructions spent on synchronization increases, while the number of instructions performing actual work remain constant. For a fixed input size, more processors share the same amount of work, thereby increasing the frequency of synchronization. In addition to this, when a thread enters a barrier, it spins there waiting for the other threads to arrive there. With more threads there is a higher chance that one of them will be "late", so more threads generally lead to more spinning on each barrier. The two Water applications evaluate forces and potentials that occur over time in a system of water molecules [38]. The Water-n2 variant uses a $O(n^2)$ algorithm for calculations. For the simulated input size, it has 12 barriers and a significant number of critical sections. Water-sp is a more efficient version of water which uses 3-d grid of cells to perform the same computation with a $O(n)$ complexity. It also has 12 barriers but a negligible number of critical sections. Both Water implementations exhibit significant synchronization contention on barriers, resulting

in large number of spin instructions. Our results differ from [24], due to two main reasons. First, our system under consideration is a CMP with much smaller round trip latencies than a distributed shared memory system(DSM). Secondly, the number of maximum processors used for simulations are 16, which represent a near future CMP, as opposed to a DSM system with processors.

The performance gains shown figure 24 are primarily due to significantly higher bus traffic caused by the transactional memory implementation we use. As mentioned in section 5.1, the transactional memory system repeatedly reads the variables used in the condition it is waiting for, increasing bus traffic. Other transactional memory implementations such as TCC [13] would likely suffer less overhead due to repeated reads in different transactions while spinning. However, repeated commits while spinning would still create overheads because commits are serialized. As a result of these considerations, we expect more advanced transactional memory systems to have better performance while spin-waiting, and EasySync performance gains would be less. However, our real goal here is to show that EasySync would not *degrade* performance significantly, and in some cases can actually improve it by a small margin. An exception to this are SMT processors, where the spinning thread competes for processor resources (ROB, physical registers, etc.) with other threads in the same core, and EasySync's stopping of the waiting thread may have a larger positive performance impact.

6.2 Optimized Code

6.2.1 Overall Results

While simulating code optimized for TM systems, the base line consisted of the optimized code running on a TM system without EasySync support. To enable this, *XWAIT* primitive was mapped on to a transaction squash and immediate retry.

Figures 25, 26, 27 show the number of instructions executed by the optimized and unoptimized code on the baseline system and EasySync system for Barnes, Raytrace and Radiosity respectively. The normalized spin instructions and idle cycles are also shown in the figure.

Barnes has been modified to use *XWAIT*, instead of while spin loop.This allows EasySync

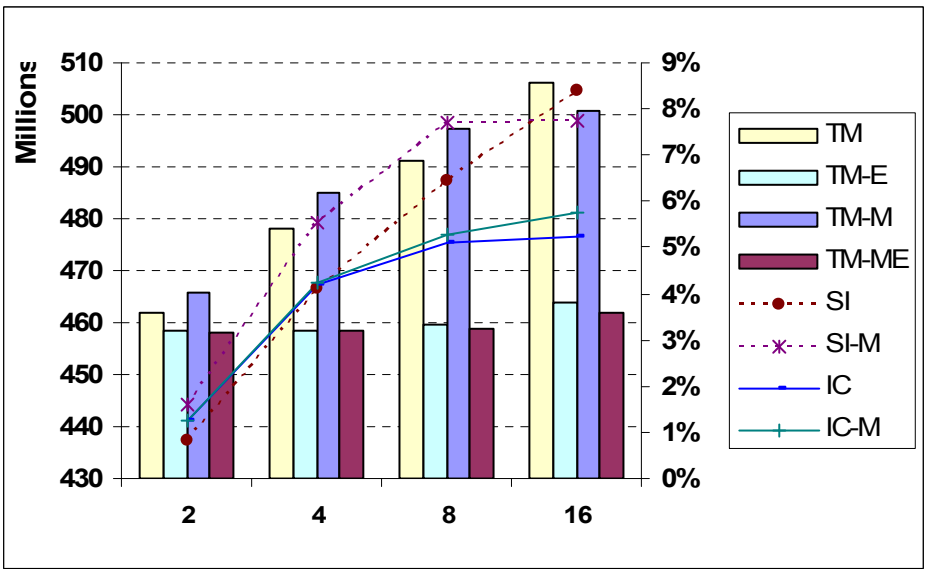


Figure 25: Modified Barnes application compared with legacy application. TM, TM-E, TM-M, TM-ME, SI, SI-M, IC, IC-M stand for legacy app. on TM ,legacy app. on TM with EasySync, modified app. on TM ,modified app on TM with EasySync, Spin instr. for legacy app., Spin instr. for modified app., Idle cycles for legacy app. and Idle cycles for modified app. respectively

to identify and eliminate the spin instructions that could not be identified and eliminated in the legacy code. The subsequent improvement in performance and efficiency can be seen in the result. Raytrace was primarily rewritten for better readability; although it also achieves minor performance improvement, the gains are within the error margin of the simulation. Radiosity is a complex application which was modified for both readability and performance. Results show marginal performance improvement.

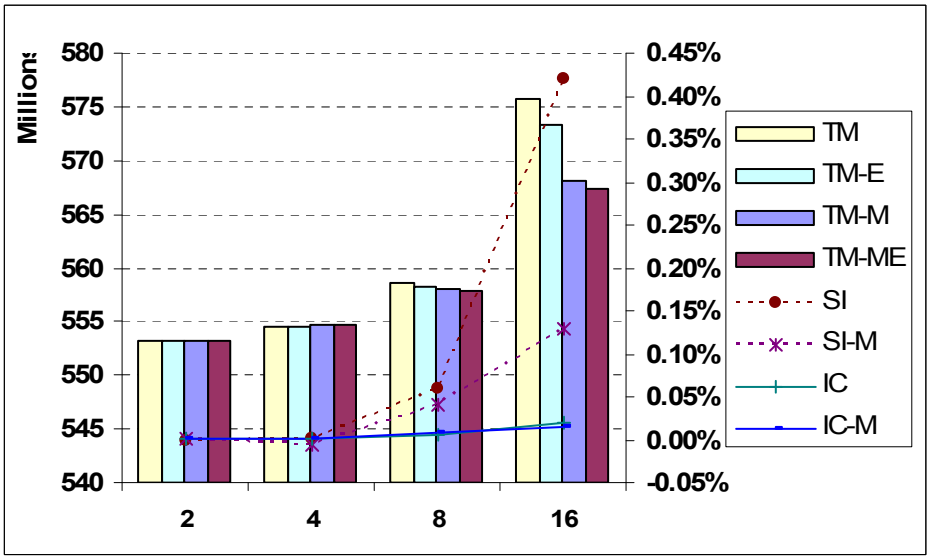


Figure 26: Modified Radiosity application compared with legacy application. TM, TM-E, TM-M, TM-ME, SI, SI-M, IC, IC-M stand for legacy app. on TM ,legacy app. on TM with EasySync, modified app. on TM ,modified app on TM with EasySync, Spin instr. for legacy app., Spin instr. for modified app., Idle cycles for legacy app. and Idle cycles for modified app. respectively

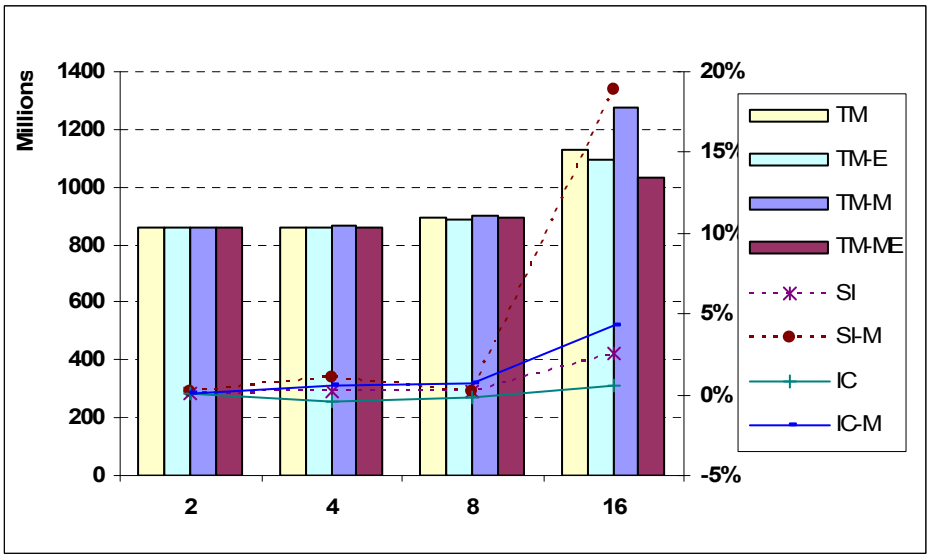


Figure 27: Modified Radiosity application compared with legacy application. TM, TM-E, TM-M, TM-ME, SI, SI-M, IC, IC-M stand for legacy app. on TM ,legacy app. on TM with EasySync, modified app. on TM ,modified app on TM with EasySync, Spin instr. for legacy app., Spin instr. for modified app., Idle cycles for legacy app. and Idle cycles for modified app. respectively

CHAPTER VII

CONCLUSIONS AND FUTURE WORK

This dissertation describes EasySync, a simple and inexpensive extension to transactional memory that allows arbitrary conditional synchronization to be expressed in a simple and composable way. Just like transactional memory eliminates the need to use locks and provides atomicity of a transaction regardless of how other code is written, EasySync eliminates the need to use conditional variables and guarantees wake-up of the waiting transaction when the real condition it is waiting for is satisfied, without the need for other code to signal that change. EasySync also allows transactional memory systems to efficiently provide lock-free and condition variable-free conditional critical regions and even more advanced synchronization primitives, such as guarded execution with arbitrary conditional or guard code.

Because EasySync informs the hardware the that a thread is waiting, it allows simple and effective optimizations, such as stopping the execution of a thread until there is a change in the condition it is waiting for. Like transactional memory, EasySync is backward-compatible with existing code, which we confirm by running unmodified Splash-2 applications linked with an EasySync-based synchronization library. The code rewritten to better exploit EasySync gives better performance than legacy applications running on a system with EasySync.

At an algorithmic level, EasySync provides interesting possibilities for parallel applications especially when code needs to be even driven. Operating systems which use Graphic user interface (GUI) based systems will in all probability significantly benefit from EasySync because of their event driven nature.Exploring this aspect presents a interesting avenue for future research.

Our experimental evaluation shows that EasySync successfully eliminates processor activity while waiting, reducing the number of executed instructions by 8.6% on average in

a 16-processor CMP. We also show that these savings increase with the number of processors, and also for applications written for transactional memory systems. Finally, EasySync imposes virtually no performance overheads, and can in fact improve performance.

APPENDIX A

MODIFIED SOURCE CODE FOR BARNES

A.1 code.C

```
/*
 * SLAVESTART: main task for each processor
 */
void SlaveStart()
{
   unsigned int ProcessId;

   /* Get unique ProcessId */
#if 0 /* Replaced code */
   LOCK(Global->CountLock);
#endif
     /* Begin atomic section */
     // asm volatile ("jal sesc_change_epoch\njal sesc_acquire_begin":::"ra");
     trans_begin();

     ProcessId = Global->current_id++;
     /* End atomic condition synchronization operation */
     // asm volatile ("jal sesc_acquire_end\njal sesc_change_epoch":::"ra");
     trans_end();
#if 0 /* Replaced code */
   UNLOCK(Global->CountLock);
#endif
   /* initialize mybodytabs */
   Local[ProcessId].mybodytab = Local[0].mybodytab + (maxmybody * ProcessId);
   /* note that every process has its own copy   */
   /* of mybodytab, which was initialized to the */
   /* beginning of the whole array by proc. 0    */
   /* before create                              */
```

```
   Local[ProcessId].mycelltab = Local[0].mycelltab + (maxmycell * ProcessId);
   Local[ProcessId].myleaftab = Local[0].myleaftab + (maxmyleaf * ProcessId);
   Local[ProcessId].tout = Local[0].tout;
   Local[ProcessId].tnow = Local[0].tnow;
   Local[ProcessId].nstep = Local[0].nstep;
   find_my_initial_bodies(bodytab, nbody, ProcessId);
   /* main loop */
   while (Local[ProcessId].tnow < tstop + 0.1 * dtime) {
      stepsystem(ProcessId);
   }
   BARRIER(Global->Barstart,NPROC);
   ENDSIM(NPROC,"barnes");
}


/*
 * STEPSYSTEM: advance N-body system one time-step.
 */
void stepsystem (ProcessId)
   unsigned int ProcessId;
{
    int i;
    real Cavg;
    bodyptr p,*pp;
    vector acc1, dacc, dvel, vel1, dpos;
    int intpow();
    unsigned int time;
    unsigned int trackstart, trackend;
    unsigned int partitionstart, partitionend;
    unsigned int treebuildstart, treebuildend;
    unsigned int forcecalcstart, forcecalcend;

    if (Local[ProcessId].nstep == 2) {
/* POSSIBLE ENHANCEMENT:  Here is where one might reset the
   statistics that one is measuring about the parallel execution */
      if (ProcessId == 0) {
```

```
    SIM_STAT_RESET();
    SIM_ON();
  }
  BARRIER(Global->Barstart,NPROC);
  STARTSIM(NPROC,"barnes");
}

if ((ProcessId == 0) && (Local[ProcessId].nstep >= 2)) {
    CLOCK(trackstart);
}

if (ProcessId == 0) {
   init_root(ProcessId);
}
else {
   Local[ProcessId].mynumcell = 0;
   Local[ProcessId].mynumleaf = 0;
}
/* start at same time */
BARRIER(Global->Barstart,NPROC);
if ((ProcessId == 0) && (Local[ProcessId].nstep >= 2)) {
    CLOCK(treebuildstart);
}
/* load bodies into tree   */
maketree(ProcessId);

if ((ProcessId == 0) && (Local[ProcessId].nstep >= 2)) {
    CLOCK(treebuildend);
    Global->treebuildtime += treebuildend - treebuildstart;
}
Housekeep(ProcessId);
Cavg = (real) Cost(Global->G_root) / (real)NPROC ;
Local[ProcessId].workMin = (int) (Cavg * ProcessId);
Local[ProcessId].workMax = (int) (Cavg * (ProcessId + 1)
                                  + (ProcessId == (NPROC - 1)));
```

```
    if ((ProcessId == 0) && (Local[ProcessId].nstep >= 2)) {
        CLOCK(partitionstart);
    }
    Local[ProcessId].mynbody = 0;
    find_my_bodies(Global->G_root, 0, BRC_FUC, ProcessId );
/*     B*RRIER(Global->Barcom,NPROC); */
    if ((ProcessId == 0) && (Local[ProcessId].nstep >= 2)) {
        CLOCK(partitionend);
        Global->partitiontime += partitionend - partitionstart;
    }
    if ((ProcessId == 0) && (Local[ProcessId].nstep >= 2)) {
        CLOCK(forcecalcstart);
    }
    ComputeForces(ProcessId);

    if ((ProcessId == 0) && (Local[ProcessId].nstep >= 2)) {
        CLOCK(forcecalcend);
        Global->forcecalctime += forcecalcend - forcecalcstart;
    }
    /* advance my bodies */
    for (pp = Local[ProcessId].mybodytab;
         pp < Local[ProcessId].mybodytab+Local[ProcessId].mynbody; pp++) {
       p = *pp;
       MULVS(dvel, Acc(p), dthf);
       ADDV(vel1, Vel(p), dvel);
       MULVS(dpos, vel1, dtime);
       ADDV(Pos(p), Pos(p), dpos);
       ADDV(Vel(p), vel1, dvel);

       for (i = 0; i < NDIM; i++) {
          if (Pos(p)[i]<Local[ProcessId].min[i]) {
             Local[ProcessId].min[i]=Pos(p)[i];
          }
          if (Pos(p)[i]>Local[ProcessId].max[i]) {
             Local[ProcessId].max[i]=Pos(p)[i] ;
```

```
            }
        }
    }
#if 0 /* Replaced code */
    LOCK(Global->CountLock);
#endif
    /* Begin atomic section */
    // asm volatile ("jal sesc_change_epoch\njal sesc_acquire_begin":::"ra");
    trans_begin();

    for (i = 0; i < NDIM; i++) {
        if (Global->min[i] > Local[ProcessId].min[i]) {
            Global->min[i] = Local[ProcessId].min[i];
        }
        if (Global->max[i] < Local[ProcessId].max[i]) {
            Global->max[i] = Local[ProcessId].max[i];
        }
    }

    /* End atomic condition synchronization operation */
    // asm volatile ("jal sesc_acquire_end\njal sesc_change_epoch":::"ra");
    trans_end();
#if 0 /* Replaced code */
    UNLOCK(Global->CountLock);
#endif
    /* bar needed to make sure that every process has computed its min */
    /* and max coordinates, and has accumulated them into the global   */
    /* min and max, before the new dimensions are computed             */
    BARRIER(Global->Barpos,NPROC);

    if ((ProcessId == 0) && (Local[ProcessId].nstep >= 2)) {
        CLOCK(trackend);
        Global->tracktime += trackend - trackstart;
    }
    if (ProcessId==0) {
```

```
      Global->rsize=0;
      SUBV(Global->max,Global->max,Global->min);
      for (i = 0; i < NDIM; i++) {
        if (Global->rsize < Global->max[i]) {
           Global->rsize = Global->max[i];
        }
      }
      ADDVS(Global->rmin,Global->min,-Global->rsize/100000.0);
      Global->rsize = 1.00002*Global->rsize;
      SETVS(Global->min,1E99);
      SETVS(Global->max,-1E99);
    }
    Local[ProcessId].nstep++;
    Local[ProcessId].tnow = Local[ProcessId].tnow + dtime;
}
```

A.2 code_io.C

```
/*
 * STOPOUTPUT: finish up after a run.
 * OUTPUT: compute diagnostics and output data.
 */

void
output (ProcessId)
   unsigned int ProcessId;
{
   int nttot, nbavg, ncavg,k;
   double cputime();
   bodyptr p, *pp;
   vector tempv1,tempv2;

   if ((Local[ProcessId].tout - 0.01 * dtime) <= Local[ProcessId].tnow) {
      Local[ProcessId].tout += dtout;
   }

   diagnostics(ProcessId);
```

```
   if (Local[ProcessId].mymtot!=0) {
#if 0 /* Replace code */
      LOCK(Global->CountLock);
#endif
      /* Begin atomic section */
      // asm volatile ("jal sesc_change_epoch\njal sesc_acquire_begin":::"ra");
                        trans_begin();

      Global->n2bcalc += Local[ProcessId].myn2bcalc;
      Global->nbccalc += Local[ProcessId].mynbccalc;
      Global->selfint += Local[ProcessId].myselfint;
      ADDM(Global->keten, Global-> keten, Local[ProcessId].myketen);
      ADDM(Global->peten, Global-> peten, Local[ProcessId].mypeten);
      for (k=0;k<3;k++) Global->etot[k] +=  Local[ProcessId].myetot[k];
      ADDV(Global->amvec, Global-> amvec, Local[ProcessId].myamvec);
      MULVS(tempv1, Global->cmphase[0],Global->mtot);
      MULVS(tempv2, Local[ProcessId].mycmphase[0], Local[ProcessId].mymtot);
      ADDV(tempv1, tempv1, tempv2);
      DIVVS(Global->cmphase[0], tempv1, Global->mtot+Local[ProcessId].mymtot);
      MULVS(tempv1, Global->cmphase[1],Global->mtot);
      MULVS(tempv2, Local[ProcessId].mycmphase[1], Local[ProcessId].mymtot);
      ADDV(tempv1, tempv1, tempv2);
      DIVVS(Global->cmphase[1], tempv1, Global->mtot+Local[ProcessId].mymtot);
      Global->mtot +=Local[ProcessId].mymtot;

      /* End atomic condition synchronization operation */
      // asm volatile ("jal sesc_acquire_end\njal sesc_change_epoch":::"ra");
      trans_end();
#if 0 /* Replaced code */
      UNLOCK(Global->CountLock);
#endif
   }
   BARRIER(Global->Baraccel,NPROC);
   if (ProcessId==0) {
```

```
      nttot = Global->n2bcalc + Global->nbccalc;
      nbavg = (int) ((real) Global->n2bcalc / (real) nbody);
      ncavg = (int) ((real) Global->nbccalc / (real) nbody);
   }
}
```

A.3 load.C

```
/*
 * MAKETREE: initialize tree structure for hack force calculation.
 */

maketree(ProcessId)
   unsigned ProcessId;
{
   bodyptr p, *pp;

   Local[ProcessId].myncell = 0;
   Local[ProcessId].mynleaf = 0;
   if (ProcessId == 0) {
      Local[ProcessId].mycelltab[Local[ProcessId].myncell++] = Global->G_root;
   }
   Local[ProcessId].Current_Root = (nodeptr) Global->G_root;
   for (pp = Local[ProcessId].mybodytab;
        pp < Local[ProcessId].mybodytab+Local[ProcessId].mynbody; pp++) {
      p = *pp;
      if (Mass(p) != 0.0) {
         Local[ProcessId].Current_Root
            = (nodeptr) loadtree(p, (cellptr) Local[ProcessId].Current_Root,
                                 ProcessId);
      }
      else {
#if 0 /* Replace code */
         LOCK(Global->io_lock);
#endif
   /* Begin atomic section */
```

```
   // asm volatile ("jal sesc_change_epoch\njal sesc_acquire_begin":::"ra");
        trans_begin();
         fprintf(stderr, "Process %d found body %d to have zero mass\n",
                 ProcessId, (int) p);
      /* End atomic condition synchronization operation */
      // asm volatile ("jal sesc_acquire_end\njal sesc_change_epoch":::"ra");
      trans_end();
#if 0 /* Replaced code */
         UNLOCK(Global->io_lock);
#endif
      }
   }
   BARRIER(Global->Bartree,NPROC);
   hackcofm( 0, ProcessId );
   BARRIER(Global->Barcom,NPROC);
}
/*
 * LOADTREE: descend tree and insert particle.
 */
nodeptr loadtree(p, root, ProcessId)
   bodyptr p;                        /* body to load into tree */
   cellptr root;
   unsigned ProcessId;
{
   int l, xq[NDIM], xp[NDIM], xor[NDIM], subindex(), flag;
   int i, j, root_level;
   bool valid_root;
   int kidIndex;
   volatile nodeptr *volatile qptr, mynode;
   cellptr c;
   leafptr le;

   intcoord(xp, Pos(p));
   valid_root = TRUE;
   for (i = 0; i < NDIM; i++) {
```

```
   xor[i] = xp[i] ^ Local[ProcessId].Root_Coords[i];
}
for (i = IMAX >> 1; i > Level(root); i >>= 1) {
   for (j = 0; j < NDIM; j++) {
      if (xor[j] & i) {
         valid_root = FALSE;
         break;
      }
   }
   if (!valid_root) {
      break;
   }
}
if (!valid_root) {
   if (root != Global->G_root) {
      root_level = Level(root);
      for (j = i; j > root_level; j >>= 1) {
         root = (cellptr) Parent(root);
      }
      valid_root = TRUE;
      for (i = IMAX >> 1; i > Level(root); i >>= 1) {
         for (j = 0; j < NDIM; j++) {
            if (xor[j] & i) {
               valid_root = FALSE;
               break;
            }
         }
         if (!valid_root) {
            printf("P%d body %d\n", ProcessId, p - bodytab);
            root = Global->G_root;
         }
      }
   }
}
root = Global->G_root;
```

```
   mynode = (nodeptr) root;
   kidIndex = subindex(xp, Level(mynode));
   qptr = &Subp(mynode)[kidIndex];
   l = Level(mynode) >> 1;
   flag = TRUE;
   while (flag) {                          /* loop descending tree     */
      if (l == 0) {
         error("not enough levels in tree\n");
      }
      if (*qptr == NULL) {
         /* lock the parent cell */
#if 0 /* Replace code */
         ALOCK(CellLock->CL, ((cellptr) mynode)->seqnum % MAXLOCK);
#endif
      /* Begin atomic section */
      // asm volatile ("jal sesc_change_epoch\njal sesc_acquire_begin":::"ra");
        trans_begin();
         if (*qptr == NULL) {
            le = InitLeaf((cellptr) mynode, ProcessId);
            Parent(p) = (nodeptr) le;
            Level(p) = l;
            ChildNum(p) = le->num_bodies;
            ChildNum(le) = kidIndex;
            Bodyp(le)[le->num_bodies++] = p;
            *qptr = (nodeptr) le;
            flag = FALSE;
         }
      /* End atomic condition synchronization operation */
      // asm volatile ("jal sesc_acquire_end\njal sesc_change_epoch":::"ra");
      trans_end();
#if 0 /* Replaced code */
         AULOCK(CellLock->CL, ((cellptr) mynode)->seqnum % MAXLOCK);
#endif
         /* unlock the parent cell */
      }
```

```
      if (flag && *qptr && (Type(*qptr) == LEAF)) {
         /*   reached a "leaf"?      */
#if 0 /* Replace code */
         ALOCK(CellLock->CL, ((cellptr) mynode)->seqnum % MAXLOCK);
#endif
      /* Begin atomic section */
      // asm volatile ("jal sesc_change_epoch\njal sesc_acquire_begin":::"ra");
                      trans_begin();

         /* lock the parent cell */
         if (Type(*qptr) == LEAF) {             /* still a "leaf"?      */
            le = (leafptr) *qptr;
            if (le->num_bodies == MAX_BODIES_PER_LEAF) {
               *qptr = (nodeptr) SubdivideLeaf(le, (cellptr) mynode, l,
                                               ProcessId);
            }
            else {
               Parent(p) = (nodeptr) le;
               Level(p) = l;
               ChildNum(p) = le->num_bodies;
               Bodyp(le)[le->num_bodies++] = p;
               flag = FALSE;
            }
         }
      /* End atomic condition synchronization operation */
      // asm volatile ("jal sesc_acquire_end\njal sesc_change_epoch":::"ra");
      trans_end();
#if 0 /* Replaced code */
         AULOCK(CellLock->CL, ((cellptr) mynode)->seqnum % MAXLOCK);
#endif
         /* unlock the node           */
      }
      if (flag) {
         mynode = *qptr;
         kidIndex = subindex(xp, l);
```

```
            qptr = &Subp(*qptr)[kidIndex];  /* move down one level  */
            l = l >> 1;                              /* and test next bit    */
         }
      }
      SETV(Local[ProcessId].Root_Coords, xp);
      return Parent((leafptr) *qptr);


/*
 * HACKCOFM: descend tree finding center-of-mass coordinates.
 */
hackcofm(nc, ProcessId)
  int nc;
  unsigned ProcessId;
{
   int i,Myindex;
   nodeptr r;
   leafptr l;
   leafptr* ll;
   bodyptr p;
   cellptr q;
   cellptr *cc;
   vector tmpv, dr;
   real drsq;
   matrix drdr, Idrsq, tmpm;
   /* get a cell using get*sub.  Cells are got in reverse of the order in */
   /* the cell array; i.e. reverse of the order in which they were created */
   /* this way, we look at child cells before parents                     */
   for (ll = Local[ProcessId].myleaftab + Local[ProcessId].mynleaf - 1;
        ll >= Local[ProcessId].myleaftab; ll--) {
      l = *ll;
      Mass(l) = 0.0;
      Cost(l) = 0;
      CLRV(Pos(l));
      for (i = 0; i < l->num_bodies; i++) {
         p = Bodyp(l)[i];
```

```
            Mass(l) += Mass(p);
            Cost(l) += Cost(p);
            MULVS(tmpv, Pos(p), Mass(p));
            ADDV(Pos(l), Pos(l), tmpv);
         }
         DIVVS(Pos(l), Pos(l), Mass(l));
#ifdef QUADPOLE
         CLRM(Quad(l));
         for (i = 0; i < l->num_bodies; i++) {
            p = Bodyp(l)[i];
            SUBV(dr, Pos(p), Pos(l));
            OUTVP(drdr, dr, dr);
            DOTVP(drsq, dr, dr);
            SETMI(Idrsq);
            MULMS(Idrsq, Idrsq, drsq);
            MULMS(tmpm, drdr, 3.0);
            SUBM(tmpm, tmpm, Idrsq);
            MULMS(tmpm, tmpm, Mass(p));
            ADDM(Quad(l), Quad(l), tmpm);
         }
#endif
         Done(l)=TRUE;
      }
      for (cc = Local[ProcessId].mycelltab+Local[ProcessId].myncell-1;
           cc >= Local[ProcessId].mycelltab; cc--) {
         q = *cc;
         Mass(q) = 0.0;
         Cost(q) = 0;
         CLRV(Pos(q));
         for (i = 0; i < NSUB; i++) {
            r = Subp(q)[i];
            if (r != NULL) {
#if 0 /* replace code */
               while(!Done(r)) {
                  /* wait */
```

```
            }
#endif
      /* Begin atomic section */
                        trans_begin();

      if (!Done(r))
                                 trans_wait();
      /* End atomic condition synchronization operation */
                        teans_end();

            Mass(q) += Mass(r);
            Cost(q) += Cost(r);
            MULVS(tmpv, Pos(r), Mass(r));
            ADDV(Pos(q), Pos(q), tmpv);
            Done(r) = FALSE;
         }
      }
      DIVVS(Pos(q), Pos(q), Mass(q));
#ifdef QUADPOLE
      CLRM(Quad(q));
      for (i = 0; i < NSUB; i++) {
         r = Subp(q)[i];
         if (r != NULL) {
            SUBV(dr, Pos(r), Pos(q));
            OUTVP(drdr, dr, dr);
            DOTVP(drsq, dr, dr);
            SETMI(Idrsq);
            MULMS(Idrsq, Idrsq, drsq);
            MULMS(tmpm, drdr, 3.0);
            SUBM(tmpm, tmpm, Idrsq);
            MULMS(tmpm, tmpm, Mass(r));
            ADDM(tmpm, tmpm, Quad(r));
            ADDM(Quad(q), Quad(q), tmpm);
         }
      }
```

```
#endif
       Done(q)=TRUE;
   }
}
```

APPENDIX B

MODIFIED SOURCE CODE FOR RADIOSITY

B.1 taskman.C

```
void process_tasks(process_id)
  unsigned process_id;
{
    Task *t ;

    t = DEQUEUE_TASK( taskqueue_id[process_id], QUEUES_VISITED, process_id ) ;

 retry_entry:
   while( t )
   {
     switch( t->task_type )
     {
        case TASK_MODELING:
           process_model( t->task.model.model, t->task.model.type, process_id ) ;
           break ;
        case TASK_BSP:
           define_patch( t->task.bsp.patch, t->task.bsp.parent, process_id ) ;
           break ;
        case TASK_FF_REFINEMENT:
           ff_refine_elements( t->task.ref.e1, t->task.ref.e2, 0, process_id ) ;
           break ;
        case TASK_RAY:
           process_rays( t->task.ray.e, process_id, process_id ) ;
           break ;
        case TASK_VISIBILITY:
           visibility_task( t->task.vis.e, t->task.vis.inter,
                t->task.vis.n_inter, t->task.vis.k, process_id ) ;
```

```
            break ;
        case TASK_RAD_AVERAGE:
            radiosity_averaging( t->task.rad.e, t->task.rad.mode, process_id ) ;
            break ;
            default:
        fprintf( stderr, "Panic:process_tasks:Illegal task type\n" );
      }

            /* Free the task */
            free_task( t, process_id ) ;

            /* Get next task */
          t = DEQUEUE_TASK( taskqueue_id[process_id],
                QUEUES_VISITED, process_id ) ;
        }


    /* Barrier. While waiting for other processors to finish, poll the task
       queues and resume processing if there is any task */

#if 0 /* replace code */
    LOCK(global->pbar_lock);
    /* Reset the barrier counter if not initialized */
    if( global->pbar_count >= n_processors )
        global->pbar_count = 0 ;

    /* Increment the counter */
    global->pbar_count++ ;
    UNLOCK(global->pbar_lock);

    /* barrier spin-wait loop */
    while( global->pbar_count < n_processors )
        {
            /* Wait for a while and then retry dequeue */
            if( _process_task_wait_loop(process_id) )
```

```
                break ;

            /* Waited for a while but other processors are still running.
               Poll the task queue again */
            t = DEQUEUE_TASK( taskqueue_id[process_id],
                QUEUES_VISITED, process_id ) ;
            if( t )
                {
                    /* Task found. Exit the barrier and work on it */
                    LOCK(global->pbar_lock);
                    global->pbar_count-- ;
                    UNLOCK(global->pbar_lock);
                    goto retry_entry ;
                }

        }
#endif
    trans_begin();
    /* Reset the barrier counter if not initialized */
    if( global->pbar_count >= n_processors )
        global->pbar_count = 0 ;

    /* Increment the counter */
    global->pbar_count++ ;
                trans_end();

                trans_begin();
                if(global->pbar_count < n_processors){

            // t = DEQUEUE_TASK( taskqueue_id[process_id],
//QUEUES_VISITED, process_id ) ;
            int qid = taskqueue_id[process_id];
            Task_Queue *tq ;
          t = 0 ;
        Task *prev ;
```

```
    int visit_count = 0 ;
  int sign = -1 ;       /* The first retry will go backward */
int offset ;

    /* Get next task */
  while( visit_count < QUEUES_VISITED )
{
    /* Select a task queue */
    tq = &global->task_queue[ qid ] ;

    /* Check the length (test-test&set) */
    if( tq->n_tasks > 0 )
        {
            /* Lock the task queue */
            // LOCK(tq->q_lock);
            /* Unlock the task queue */
            // UNLOCK(tq->q_lock);
            break ;
        }

    /* Update visit count */
    visit_count++ ;

    /* Compute next taskqueue ID */
    offset = (sign > 0)? visit_count : -visit_count ;
    sign = -sign ;

    qid += offset ;
    if( qid < 0 )
        qid += n_taskqueues ;
    else if( qid >= n_taskqueues )
        qid -= n_taskqueues ;
}
```

```
if( tq->top )
  {
                                    trans_exit();
    if( qid == taskqueue_id[process_id] )
      {
        t = tq->top ;
        tq->top = t->next ;
        if( tq->top == 0 )
          tq->tail = 0 ;
        tq->n_tasks-- ;
            }
    else
      {
        /* Get tail */
        for( prev = 0, t = tq->top ; t->next ;
          prev = t, t = t->next ) ;

        if( prev == 0 )
          tq->top = 0 ;
        else
          prev->next = 0 ;
        tq->tail = prev ;
        tq->n_tasks-- ;
      }
  }
else
  trans_wait();

                if(t){
                /* Task found. Exit the barrier and work on it */
    global->pbar_count-- ;
                          trans_end();
        goto retry_entry ;
                }
          }
```

```
    trans_end();

    BARRIER(global->barrier, n_processors);
}
```

APPENDIX C

MODIFIED SOURCE CODE FOR RAYTRACE

C.1 memory.C

```
VOID    *GlobalRealloc(p, size)
VOID    *p;
UINT    size;
        {
        UINT            oldsize;
        UINT            newsize;
        UINT            totsize;
        VOID    huge    *q;
        UINT    huge    *r;
        UINT    huge    *s;
        NODE    huge    *pn;
        NODE    huge    *prev;
        NODE    huge    *curr;
        NODE    huge    *next;
        NODE    huge    *node;

        if (!size)
                {
                GlobalFree(p);
                return (NULL);
                }

        if (!p)
                return (GlobalMalloc(size, "GlobalRealloc"));


        pn = NODE_ADD(p, -nodesize);            /* Adjust ptr back to arena. */
```

```
        if (pn->cksm != CKSM)
                {
                fprintf(stderr, "GlobalRealloc: Attempted to realloc node with
invalid checksum.\n");
                exit(1);
                }

        if (pn->free)
                {
                fprintf(stderr, "GlobalRealloc: Attempted to realloc an
unallocated node.\n");
                exit(1);
                }


        newsize = ROUND_UP(size);
        oldsize = pn->size;


        /*
         *      If new size is less than current node size, truncate the node
         *      and return end to free list.
         */

        if (newsize <= oldsize)
                {
                if (oldsize - newsize < THRESHOLD)
                        return (p);

                pn->size    = newsize;

                next        = NODE_ADD(p, newsize);
                next->size  = oldsize - nodesize - newsize;
                next->next  = NULL;
                next->free  = FALSE;
```

```
                next->cksm  = CKSM;
                next        = NODE_ADD(next, nodesize);

                GlobalFree(next);
                return (p);
                }


        /*
         *      New size is bigger than current node.  Try to expand next node
         *      in list.
         */

        next    = NODE_ADD(p, oldsize);
        totsize = oldsize + nodesize + next->size;

        // LOCK(gm->memlock)
        if (next < endmem && next->free && totsize >= newsize)
                {
                /* Find next in free list. */
                trans_begin();
                prev = NULL;
                curr = gm->freelist;

                while (curr && curr < next && curr < endmem)
                        {
                        prev = curr;
                        curr = curr->next;
                        }

                if (curr != next)
                        {
                        fprintf(stderr, "GlobalRealloc: Could not find next node
in free list.\n");
                        exit(1);
```

```
        }

if (totsize - newsize < THRESHOLD)
        {
        /* Just remove next from free list. */

        if (!prev)
                gm->freelist = next->next;
        else
                prev->next   = next->next;

        next->next = NULL;
        next->free = FALSE;
        pn->size   = totsize;

        // UNLOCK(gm->memlock)
        // return (p);
        }
else
        {
        /* Remove next from free list while adding node. */

        node       = NODE_ADD(p, newsize);
        node->next = next->next;
        node->size = totsize - nodesize - newsize;
        node->free = TRUE;
        node->cksm = CKSM;

        if (!prev)
                gm->freelist = node;
        else
                prev->next   = node;

        next->next = NULL;
        next->free = FALSE;
```

```
                pn->size   = newsize;

                // UNLOCK(gm->memlock)
                // return (p);
                }
        trans_end();
        return (p);
        }
/*
 *      New size is bigger than current node, but next node in list
 *      could not be expanded.  Try to allocate new node and move data
 *      to new location.
 */

//UNLOCK(gm->memlock)

s = q = GlobalMalloc(newsize, "GlobalRealloc");
if (!q)
        return (NULL);

r = (UINT huge *)p;
oldsize >>= 2;

while (oldsize--)
        *s++ = *r++;

GlobalFree(p);
return (q);
}
```

REFERENCES

[1] "Using spin-loops on intel pentium 4 processor and intel xeon processor." Application note, Intel Corporation, May 2001.

[2] *IA-32 Intel Architecture Software Developer's Manual, Volume 3A: System Programming Guide*, 2004.

[3] "Microsoft developer network online documentation." http://msdn.microsoft.com/library/ default.asp?url=/library/en-us/winui/winui/windowsuserinterface/windowing/windowproce dures/aboutwindowprocedures.asp, Mar 2006.

[4] "Splash2 application suite source code." http://www-flash.stanford.edu/apps/SPLASH/, Mar 2006.

[5] Adve, S. V. and Gharachorloo, K., "Shared memory consistency models: A tutorial," *IEEE Computer*, vol. 29, no. 12, pp. 66–76, 1996.

[6] Ananian, C. S., Asanović, K., Kuszmaul, B. C., Leiserson, C. E., and Lie, S., "Unbounded transactional memory," in *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA'05)*, (San Franscisco, California), pp. 316–327, Feb. 2005.

[7] Choi, S.-E. and Lewis, E. C., "A study of common pitfalls in simple multi-threaded programs," in *In Proceedings of the Thirty-First ACM SIGCSE Technical Symposium on Computer Science Education*, pp. 325 – 329, March 2000.

[8] Cintra, M., Martínez, J. F., and Torrellas, J., "Architectural support for scalable speculative parallelization in shared-memory multiprocessors," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pp. 13–24, ACM Press, 2000.

[9] Dijkstra, E. W., "Guarded commands, nondeterminacy and formal derivation of programs," *Communications of the ACM*, vol. 18, no. 8, pp. 453–457, 1975.

[10] Emer, J. S., "Simultaneous multithreading: Multiplying alpha's performance," Oct. 1999.

[11] Foster, I., *Designing and Building Parallel Programs*. Addison-Wesley, 1995.

[12] Gray, J., "The transaction concept: Virtues and limitations," in *Very Large Data Bases, 7th International Conference, Proceedings*, pp. 144–154, IEEE Computer Society, Sept. 1981.

[13] Hammond, L., Wong, V., Chen, M., Carlstrom, B. D., Davis, J. D., Hertzberg, B., Prabhu, M. K., Wijaya, H., Kozyrakis, C., and Olukotun, K., "Transactional Memory Coherence and Consistency," in *Proceedings of the 31st International Symposium on Computer Architecture (ISCA)*, pp. 102–113, 2004.

[14] Hansen, P. B., "Distributed processes: a concurrent programming concept," *Communications of the ACM*, vol. 21, no. 11, pp. 934–941, 1978.

[15] Hansen, P. B., "Edison – a multiprocessor language," *Software – Practice and Experience*, vol. 11, no. 4, pp. 325–361, 1981.

[16] Harris, T. and Fraser, K., "Language support for lightweight transactions," in *Proceedings of the 18th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*, pp. 388–402, ACM Press, 2003.

[17] Herlihy, M. and Moss, J. E. B., "Transactional memory: Architectural support for lock-free data structures," in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pp. 289–300, ACM Press, 1993.

[18] Hoare, C. A. R., "Towards a theory of parallel programming," in *Operating Systems Techniques* (Hoare, C. A. R. and Perrott, R. H., eds.), no. 9 in A.P.I.C. Studies in Data Processing, pp. 61–71, Academic Press, 1972.

[19] Hoare, C. A. R., "Communicating sequential processes," *Communications of the ACM*, vol. 21, no. 8, pp. 666–677, 1978.

[20] Kapali P. Eswaran, Jim Gray, R. A. L. and Traiger, I. L., "The notions of consistency and predicate locks in a database system," *Communications of the ACM*, vol. 19, no. 11, pp. 624–633, 1976.

[21] Kohavi, Z., *Switching and Finite Automata Theory: Computer Science Series*. McGraw-Hill Higher Education, 1990.

[22] Lamport, L., "How to Make a Multiprocessor Computer That correctly Executes Multiprocess Programs," *IEEE Transactions on Computers*, vol. C-28, no. 9, pp. 690–691, 1979.

[23] Li, J., Martez, J. F., and Huang, M. C., "The thrifty barrier: Energy-aware synchronization in shared-memory multiprocessors," in *Proceedings of the International Symposium on High-Performance Computer Architecture*, Feb. 2004.

[24] Li, J., Martinez, J. F., and Huang, M. C., "The Thrifty Barrier: Energy-Aware Synchronization in Shared-Memory Multiprocessors," in *Proceedings of Proceedings of the Tenth International Symposium on High Performance Computer Architecture*, 2004.

[25] Lomet, D. B., "Process structuring, synchronization, and recovery using atomic actions," in *Proceedings of the ACM Conference on Language Design for Reliable Software*, pp. 128–137, 1977.

[26] Martínez, J. F. and Torrellas, J., "Speculative Synchronization: Programmability and Performance for Parallel Codes," *IEEE Micro*, vol. 23, no. 6, pp. 126–134, 2003.

[27] Moore, K. E., Bobba, J., Moravan, M. J., Hill, M. D., and Wood, D. A., "Logtm: Log-based transactional memory," in *Proceedings of the 12th International Symposium on High-Performance Computer Architecture (HPCA-12)*, February 2006.

[28] Ousterhout, J., "Why Threads Are A Bad Idea (for most purposes)," *1996 USENIX Technical Conference (Invited Talk)*, Jan. 1996.

[29] Prvulovic, M., Garzaran, M. J., Rauchwerger, L., and Torrellas, J., "Removing Architectural Bottlenecks to the Scalability of Speculative Parallelization," in *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pp. 204–215, July 2001.

[30] Rajwar, R. and Goodman, J. R., "Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution," in *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 294–305, IEEE Computer Society, 2001.

[31] Rajwar, R. and Goodman, J. R., "Transactional Lock-Free Execution of Lock-Based Programs," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 5–17, ACM Press, 2002.

[32] Rajwar, R., Herlihy, M., and Lai, K., "Virtualizing transactional memory," in *ISCA '05: Proceedings of the 32nd Annual International Symposium on Computer Architecture*, (Washington, DC, USA), pp. 494–505, IEEE Computer Society, 2005.

[33] Renau, J. and others, "SESC," *http://sesc.sourceforge.net*, Mar 2006.

[34] Richard L. Sites, e., *Alpha Architecture Reference Manual*, 1992.

[35] Scott, M. L., "Language support for loosely coupled distributed programs," *IEEE Transactions on Software Engineering*, vol. 13, no. 1, pp. 88–103, 1987.

[36] Singh, J. P., Hennessy, J. L., and Gupta, A., "Implications of hierarchical n-body methods for multiprocessor architectures," *ACM Trans. Comput. Syst.*, vol. 13, no. 2, pp. 141–202, 1995.

[37] Tullsen, D. M., Eggers, S., and Levy, H. M., "Simultaneous multithreading: Maximizing on-chip parallelism," in *Proceedings of the 22th Annual International Symposium on Computer Architecture*, 1995.

[38] Woo, S. C., Ohara, M., Torrie, E., Singh, J. P., and Gupta, A., "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *Proceedings of the 22nd International Symposium on Computer Architecture (ISCA)*, pp. 24–36, ACM Press, 1995.